C 语言深度解剖

-------------------解开程序员面试笔试的秘密

以含金量勇敢挑战国内外同类书籍

陈正冲　　编著
石　虎　　审阅

版权申明

写在前言前面的话

最近面试了一些人，包括应届本科、硕士和工作多年的程序员，在问到 C 语言相关的问题的时候，总是没几个人能完全答上我的问题。甚至一些工作多年，简历上写着“最得意的语言是 C 语言”，“对 C 有很深的研究”，“精通 C 语言”的人也答不完全我的问题，甚至有个别人我问的问题一个都答不上。于是我就想起了我去年闲的使用写的这本小册子。

这本小册子已经在我电脑里睡了一年大觉了。并非没有出版社愿意出版，而是几个大的出版社都认为书写得不错，但太薄，利润太低，所以要求我加厚到 300 页以上。我拒绝加厚，并为此和几个出版社僵持了一年多。我认为经典的东西一定要精炼，不要废话。这次由于面试别人，所以终于记起了我还写过这么一本小册子。想了想，还是决定挂到网上免费让大家看得了。并为此专门为本书开了个博客，以方便和读者交流。博客地址：http://blog.csdn.net/dissection_c

作者简介：

陈正冲：湖南沅江人，毕业于长春光学精密机械学院（长春理工大学）数学系。目前从事嵌入式软件开发和管理方面的工作。

石虎：湖南沅江人，毕业于吉林大学计算机系。目前为大连交通大学计算机系讲师。

前　言

我遇到过很多程序员和计算机系毕业的学生，也给很多程序员和计算机系毕业的学生讲解过《高级 C 语言程序设计》。每期班开课前，我总会问学生：你感觉 C 语言学得怎么样？难吗？指针明白吗？数组呢？内存管理呢？往往学生回答说：感觉还可以，C 语言不难，指针很明白，数组很简单，内存管理也不难。一般我会再问一个问题：通过这个班的学习，你想达到什么程度？很多学生回答：精通 C 语言。我告诉他们：我很无奈，也很无语。因为我完全在和一群业余者或者是 C 语言爱好者在对话。你们大学的计算机教育根本就是在浪费你们的时间，念了几年大学，连 C 语言的门都没摸着。现在大多数学校计算机系都开了 C、C++、Java、C#等等语言，好像什么都学了，但是什么都不会，更可悲的是有些大学居然取消了 C 语言课程，认为其过时了。我个人的观点是“十鸟在林，不如一鸟在手”，真正把 C 语言整明白了再学别的语言也很简单，如果 C 语言都没整明白，别的语言学得再好也是花架子，因为你并不了解底层是怎么回事。当然我也从来不认为一个没学过汇编的人能真正掌握 C 语言的真谛。我个人一直认为，普通人用 C 语言在 3 年之下，一般来说，还没掌握 C 语言；5 年之下，一般来说还没熟悉 C 语言；10 年之下，谈不上精通。所以，我告诉我的学生：听完我的课，远达不到精通的目标，熟悉也达不到，掌握也达不到。那能达到什么目标？-----领你们进入 C 语言的大门。入门之后的造化如何在于你们自己。不过我可以告诉你们一条不是捷径的捷径：把一个键盘的 F10 或 F11 按坏，当然不能是垃圾键盘。

往往讲到这里，学生眼里总是透露着疑虑。C 语言有这么难吗？我的回答是：不难。但你就是用不明白。学生说：以前大学老师讲 C 语言，我学得很好。老师讲的都能听懂，考试也很好。平时练习感觉自己还不错，工作也很轻松找到了。我告诉学生：听明白，看明白不代表你懂了，你懂了不代表你会用了，你会用了不代表你能用明白，你能用明白不代表你真正懂了！什么时候表明你真正懂了呢？你站在我这来，把问题给下面的同学讲明白，学生都听明白了，说明你真正懂了。否则，你就没真正懂，这是检验懂没懂的唯一标准。冰山大家都没见过，但总听过或是电影里看过吧？如果你连《泰坦尼克》都没看过，那你也算个人物（开个玩笑）。《泰坦尼克》里的冰山给泰坦尼克造成了巨大的损失。你们都是理工科的，应该明白冰山在水面上的部分只是总个冰山的 1/8。我现在就告诉你们，C 语言就是这座冰山。你们现在仅仅是摸到了水面上的部分，甚至根本不知道水面下的部分。我希望通过我的讲解，让你们摸到水面下的部分，让你们知道 C 语言到底是什么样子。

从现在开始，除非在特殊情况下，不允许用 printf 这个函数。为什么呢？很多学生写完代码，直接用 printf 打印出来，发现结果不对。然后就举手问我:老师，我的结果为什么不对啊？连调试的意识都没有！大多数学生根本就不会调试，不会看变量的值，内存的值。只知道 printf 出来结果不对，却不知道为什么不对，怎么解决。这种情况还算好的。往往很多时候 printf 出来的结果是对的，然后呢，学生也理所当然的认为程序没有问题。是这样吗？往往不是，往后看，你能看到例子的。永远给我记住一点：结果对，并不代表程序真正没有问题。所以，以后尽量不要用 printf 函数，要去看变量的值，内存的值。当然，在我们目前的编译器里，变量的值，内存的值对了就代表你程序没问题吗？也不是，往后，你也会看到例子的。

这个时候呢，学生往往会莫名其妙。这个老师有问题吧。大学里我们老师都教我们怎么用 printf，告诉我们要经常用 printf。这也恰恰是大学教育失败的地方之一。很多大学老师根本就没真正用 C 语言写过几行代码，更别说教学生调试代码了。不调试代码，不按 F10 或F11，水平永远也无法提上来，所以，要想学好一门编程语言，最好的办法就是多调试。你去一个软件公司转转，去看人家的键盘，如果发现键盘上的 F10 或 F11 铮亮铮亮，毫无疑问，此机的主人曾经或现在是开发人员（这里仅指写代码的，不上升到架构设计类的开发人员），

否则，必是非开发人员。

非常有必要申明，本人并非什么学者或是专家，但本人是数学系毕业，所以对理论方面比较擅长。讲解的时候会举很多例子来尽量使学生明白这个知识点，至于这些例子是否恰当则是见仁见智的问题了。但是一条，长期的数学训练使得本人思维比较严谨，讲解一些知识点尤其是一些概念性原理性的东西时会抠的很细、很严，这一点相信读者会体会得到的。本书是我平时讲解 C 语言的一些心得和经验，其中有很多我个人的见解或看法。经过多期培训班的实践，发现这样讲解得比较透彻，学生听得明白。很多学生听完课后告诉我：我有生以来听课从来都没有听得这么透彻，这么明白过。也有业余班的学生甚至辞掉本职工作来听我的课的。

当然，关于 C 语言的这么多经验和心得的积累并非我一人之力。借用一句名言：我只不过是站在巨人的肩膀上而已。给学生做培训的时候我参考得比较多的书有：Kernighan & Ritchie 的《The C Programming Language》；Linden 的《Expert C Programming》； Andrew & Koening《C Traps and Pitfalls》； Steve Maguire 的《Write Clean Code》；Steve McConnell 的《Code Complete. Second Edition》；林锐的《高质量 C++/C 编程指南》。这些书都是经典之作，但却都有着各自的缺陷。读者往往需要同时阅读这些书才能深刻的掌握某一知识点。我的讲课的试图时候融各家之长，再加上我个人的见解传授给学生。还好，学生反映还可以，至少还没有出乱子。这些书饱含着作者的智慧，每读一遍都有不同的收获，我希望读者能读上十遍。另外，在编写本书时也参考了网上一些无名高手的文章，这些高手的文章见解深刻，使我受益匪浅。这里要感谢这些大师们，如果不是他们，肯怕我的 C 语言的水平也仅仅是入门而已。

学习 C 语言，这几本书如果真正啃透了，水平不会差到哪。与其说本书是我授课的经验与心得，不如说本书是我对这些大师们智慧的解读。本书并不是从头到尾讲解 C 语言的基础知识，所以，本书并不适用于 C 语言零基础的人。本书的知识要比一般的 C 语言书说讲的深的多，其中有很多问题是各大公司的面试或笔试题。所以本书的读者应该是中国广大的计算机系的学生和初级程序员。如果本书上面的问题能真正明白 80%，作为一个应届毕业生，肯怕没有一家大公司会拒绝你。当然，书内很多知识也值得计算机教师或是中高级程序员参考。尤其书内的一些例子或比方，如果能被广大教师用于课堂，我想对学生来说是件非常好的事情。有人说电影是一门遗憾的艺术，因为在编辑完成之后总能或多或少的发现一些本来可以做得更好的缺陷。讲课同样也如此，每次讲完课之后总能发现自己某些地方或是没有讲到，或是没能讲透彻或是忘了举一个轻浅的例子等等。整理本书的过程也是，为了尽量精炼，总是犹豫一些东西的去留。限于作者水平，书中难免有些遗漏甚至错误，希望各位读者能予指教。作者 Mail:dissection_c@163.com.

陈正冲

2008 年 6 月 23 日

目　　录

第一章 关键字

每次讲关键字之前，我总是问学生：C 语言有多少个关键字？sizeof 怎么用？它是函数吗？有些学生不知道 C 语言有多少个关键字，大多数学生往往告诉我 sizeof 是函数，因为它后面跟着一对括号。当投影仪把这 32 个关键字投到幕布上时，很多学生表情惊讶。有些关键字从来没见过，有的惊讶 C 语言关键字竟有 32 个之多。更有甚者，说大学老师告诉他们 sizeof 是函数，没想到它居然是关键字！由此可想而知，大学的计算机教育是多么失败！

表(1.1)C 语言标准定义的 32 个关键字

关键字	意 义
auto	声明自动变量，缺省时编译器一般默认为 auto
int	声明整型变量
double	声明双精度变量
long	声明长整型变量
char	声明字符型变量
float	声明浮点型变量
short	声明短整型变量
signed	声明有符号类型变量
unsigned	声明无符号类型变量
struct	声明结构体变量
union	声明联合数据类型
enum	声明枚举类型
static	声明静态变量
switch	用于开关语句
case	开关语句分支
default	开关语句中的“其他”分支
break	跳出当前循环
register	声明寄存器变量
const	声明只读变量
volatile	说明变量在程序执行中可被隐含地改变
typedef	用以给数据类型取别名(当然还有其他作用)

extern	声明变量是在其他文件正声明(也可以看做是引用变量)
return	子程序返回语句(可以带参数，也可不带参数)
void	声明函数无返回值或无参数，声明空类型指针
continue	结束当前循环，开始下一轮循环
do	循环语句的循环体
while	循环语句的循环条件
if	条件语句
else	条件语句否定分支(与 if 连用)
for	一种循环语句(可意会不可言传)
goto	无条件跳转语句
sizeof	计算对象所占内存空间大小

下面的篇幅就一一讲解这些关键字。但在讲解之前先明确两个概念：

什么是定义？什么是声明？它们有何区别?

举个例子：

A)int i；

B)extern int i；(关于 extern，后面解释)

哪个是定义？哪个是声明？或者都是定义或者都是声明？我所教过的学生几乎没有一人能回答上这个问题。这个十分重要的概念在大学里从来没有被提起过！

什么是定义：所谓的定义就是(编译器)创建一个对象，为这个对象分配一块内存并给它取上一个名字，这个名字就是我们经常所说的变量名或对象名。但注意，这个名字一旦和这块内存匹配起来(可以想象是这个名字嫁给了这块空间，没有要彩礼啊。^_^)，它们就同生共死，终生不离不弃。并且这块内存的位置也不能被改变。一个变量或对象在一定的区域内（比如函数内，全局等）只能被定义一次，如果定义多次，编译器会提示你重复定义同一个变量或对象。

什么是声明：有两重含义，如下：

第一重含义：告诉编译器，这个名字已经匹配到一块内存上了(伊人已嫁，吾将何去何从？何以解忧，唯有稀粥)，下面的代码用到变量或对象是在别的地方定义的。声明可以出现多次。

第二重含义：告诉编译器，我这个名字我先预定了，别的地方再也不能用它来作为变量名或对象名。比如你在图书馆自习室的某个座位上放了一本书，表明这个座位已经有人预订，别人再也不允许使用这个座位。其实这个时候你本人并没有坐在这个座位上。这种声明最典型的例子就是函数参数的声明，例如：

void fun(int i, char c);

好，这样一解释，我们可以很清楚的判断:A)是定义；B)是声明。

那他们的区别也很清晰了。记住，**定义声明最重要的区别：定义创建了对象并为这个**

对象分配了内存，声明没有分配内存(一个抱伊人，一个喝稀粥。^_^)。

1.1，最宽恒大量的关键字----auto

auto：它很宽恒大量的，你就当它不存在吧。编译器在默认的缺省情况下，所有变量都是 auto 的。

1.2，最快的关键字---- register

register：这个关键字请求编译器**尽可能**的将变量存在 CPU 内部寄存器中而不是通过内存寻址访问以提高效率。注意是**尽可能，不是绝对**。你想想，一个 CPU 的寄存器也就那么几个或几十个，你要是定义了很多很多 register 变量，它累死也可能不能全部把这些变量放入寄存器吧，轮也可能轮不到你。

1.2.1，皇帝身边的小太监----寄存器

不知道什么是寄存器？那见过太监没有？没有？其实我也没有。没见过不要紧，见过就麻烦大了。^_^，大家都看过古装戏，那些皇帝们要阅读奏章的时候，大臣总是先将奏章交给皇帝旁边的小太监，小太监呢再交给皇帝同志处理。这个小太监只是个中转站，并无别的功能。

好，那我们再联想到我们的 CPU。CPU 不就是我们的皇帝同志么？大臣就相当于我们的内存，数据从他这拿出来。那小太监就是我们的寄存器了（这里先不考虑 CPU 的高速缓存区）。数据从内存里拿出来先放到寄存器，然后 CPU 再从寄存器里读取数据来处理，处理完后同样把数据通过寄存器存放到内存里，CPU 不直接和内存打交道。这里要说明的一点是:小太监是主动的从大臣手里接过奏章，然后主动的交给皇帝同志，但寄存器没这么自觉，它从不主动干什么事。一个皇帝可能有好些小太监，那么一个 CPU 也可以有很多寄存器，不同型号的 CPU 拥有寄存器的数量不一样。

为啥要这么麻烦啊？速度！就是因为速度。寄存器其实就是一块一块小的存储空间，只不过其存取速度要比内存快得多。进水楼台先得月嘛，它离 CPU 很近，CPU 一伸手就拿到数据了，比在那么大的一块内存里去寻找某个地址上的数据是不是快多了？那有人问既然它速度那么快，那我们的内存硬盘都改成寄存器得了呗。我要说的是：你真有钱！

1.2.2，使用 register 修饰符的注意点

虽然寄存器的速度非常快，但是使用 register 修饰符也有些限制的：register 变量必须是能被 CPU 寄存器所接受的类型。意味着 register 变量必须是一个单个的值，并且其长度应小于或等于整型的长度。 而且 register 变量可能不存放在内存中，所以不能用取址运算符“&”来获取 register 变量的地址。

1.3，最名不符实的关键字----static

不要误以为关键字 static 很安静，其实它一点也不安静。这个关键字在 C 语言里主要有两个作用，C++对它进行了扩展。

1.3.1，修饰变量

第一个作用：修饰变量。变量又分为局部和全局变量，但它们都存在内存的静态区。

静态全局变量，作用域**仅限于变量被定义的文件中**，其他文件即使用 extern 声明也没法使用他。准确地说作用域是从定义之处开始，到文件结尾处结束，在定义之处前面的那些代码行也不能使用它。想要使用就得在前面再加 extern ***。恶心吧？要想不恶心，很简单，直接在文件顶端定义不就得了。

静态局部变量，在函数体里面定义的，就只能在这个函数里用了，同一个文档中的其他函数也用不了。由于被 static 修饰的变量总是存在内存的静态区，所以即使这个函数运行结束，这个静态变量的值还是不会被销毁，函数下次使用时仍然能用到这个值。

```
static int   j;
void fun1（void）
{
    static int i = 0;
    i ++;
}
void fun2（void）
{
    j = 0;
    j++;
}
int main()
{
    for(k=0; k<10; k++)
    {
        fun1();
        fun2();
    }
    return 0;
```

}

i 和 j 的值分别是什么,为什么？

1.3.2，修饰函数

第二个作用：修饰函数。函数前加 static 使得函数成为静态函数。但此处“static”的含义不是指存储方式，而是指对函数的作用域仅局限于本文件(所以又称内部函数)。使用内部函数的好处是：不同的人编写不同的函数时，不用担心自己定义的函数，是否会与其它文件中的函数同名。

关键字 static 有着不寻常的历史。起初，在 C 中引入关键字 static 是为了表示退出一个块后仍然存在的局部变量。随后，static 在 C 中有了第二种含义：用来表示不能被其它文件访问的全局变量和函数。为了避免引入新的关键字，所以仍使用 static 关键字来表示这第二种含义。

当然，C++里对 static 赋予了第三个作用，这里先不讨论，有兴趣的可以找相关资料研究。

1.4，基本数据类型----short、int、long、char、float、double

C 语言包含的数据类型如下图所示：

- C 数据类型
  - 基本类型
    - 数值类型
      - 整型
        - 短整型 short
        - 整型 int
        - 长整型 long
      - 浮点型
        - 单精度型 float
        - 双精度型 double
    - 字符类型 char
  - 构造类型
    - 数组
    - 结构体 struct
    - 共用体 union
    - 枚举类型 enum
  - 指针类型
  - 空类型 void

1.4.1，数据类型与“模子”

short、int、long、char、float、double 这六个关键字代表 C 语言里的六种基本数据类型。怎么去理解它们呢? 举个例子：见过藕煤球的那个东西吧？(没见过？煤球总见过吧)。那个东西叫藕煤器，拿着它在和好的煤堆里这么一咔，一个煤球出来了。半径 12cm，12 个孔。不同型号的藕煤器咔出来的煤球大小不一样，孔数也不一样。这个藕煤器其实就是个**模子**。

现在我们联想一下，short、int、long、char、float、double 这六个东东是不是很像不同类型的藕煤器啊？拿着它们在内存上咔咔咔，不同大小的内存就分配好了，当然别忘了给它们取个好听的名字。在 32 位的系统上 short 咔出来的内存大小是 2 个 byte；int 咔出来的内存大小是 4 个 byte；long 咔出来的内存大小是 4 个 byte；float 咔出来的内存大小是 4 个 byte；double 咔出来的内存大小是 8 个 byte；char 咔出来的内存大小是 1 个 byte。（注意这里指一般情况，可能不同的平台还会有所不同，具体平台可以用 sizeof 关键字测试一下）

很简单吧？咔咔咔很爽吧？是很简单，也确实很爽，但问题就是你咔出来这么多内存块，你总不能给他取名字叫做 x1,x2,x3,x4,x5…或者长江 1 号，长江 2 号…吧。它们长得这么像(不是你家的老大，老二，老三…)，过一阵子你就会忘了到底哪个名字和哪个内存块匹配了(到底谁嫁给谁了啊？^_^)。所以呢，给他们取一个好的名字绝对重要。下面我们就来研究研究取什么样的名字好。

1.4.2，变量的命名规则

一般规则：

【规则 1-1】命名应当直观且可以拼读，可望文知意，便于记忆和阅读。

标识符最好采用英文单词或其组合，不允许使用拼音。程序中的英文单词一般不要太复杂，用词应当准确。

【规则 1-2】命名的长度应当符合“min-length && max-information”原则。

C 是一种简洁的语言, 命名也应该是简洁的。例如变量名 MaxVal 就比 MaxValueUntilOverflow 好用。标识符的长度一般不要过长，较长的单词可通过去掉“元音”形成缩写。

另外，英文词尽量不缩写，特别是非常用专业名词，如果有缩写，在同一系统中对同一单词必须使用相同的表示法，并且注明其意思。

【规则 1-3】当标识符由多个词组成时，每个词的第一个字母大写，其余全部小写。比如：

```
int   CurrentVal;
```

这样的名字看起来比较清晰，远比一长串字符好得多。

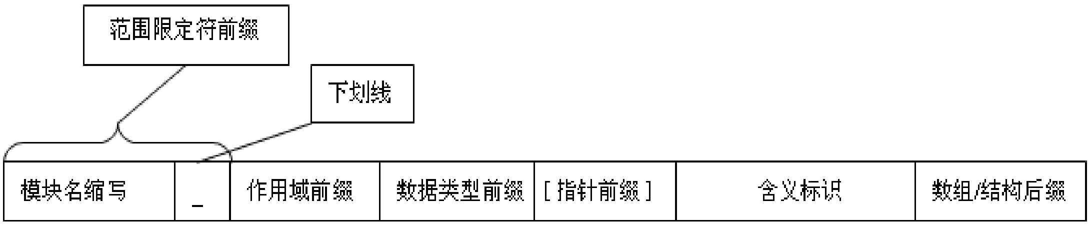
【规则 1-4】尽量避免名字中出现数字编号，如 Value1,Value2 等，除非逻辑上的确需要编号。比如驱动开发时为管脚命名，非编号名字反而不好。

初学者总是喜欢用带编号的变量名或函数名，这样子看上去很简单方便，但其实是一颗颗定时炸弹。这个习惯初学者一定要改过来。

【规则 1-5】对在多个文件之间共同使用的全局变量或函数要加范围限定符(建议使用模块名(缩写)作为范围限定符)。(GUI_ ，etc)

标识符的命名规则：

【规则 1-6】标识符名分为两部分：规范标识符前缀(后缀) + 含义标识 。非全局变量可以不用使用范围限定符前缀。



【规则 1-7】作用域前缀命名规则。

No.	标识符类型	作用域前缀
1	Global Variable	g
2	File Static Variable(native)	n
3	Function Static Variable	f
4	Auto Variable	a
5	Global Function	g
6	Static Function	n

【规则 1-8】数据类型前缀命名规则。

No.	Prefix	Suffix	Data Type	Example	Remark
1	bt		bit	Bit btVariable;	
2	b		boolean	boolean bVariable;	
3	c		char	char cVariable;	
4	i		int	int iVariable;	
5	s		short[int]	short[int] sVariable;	
6	l		long[int]	long[int] lVariable;	
7	u		unsigned[int]	unsigned[int] uiVariable;	
8	d		double	double dVariable;	
9	f		float	float fVariable;	

10	p		pointer	void *vpVariable;	指针前缀
11	v		void	void vVariable;	
13	st		enum	enum A stVariable;	
14	st		struct	struct A stVariable;	
15	st		union	union A stVariable;	
16	fp		function point	void(* fpGetModeFuncList_a[])(void)	
17		_a	array of	char cVariable_a[TABLE_MAX];	
18		_st _pst	typedef enum/struct/union	typedef struct SM_EventOpt { unsigned char unsigned int char }SM_EventOpt_st,*SM_EventOpt_pst;	当自定义结构数据类型时使用_st 后缀； 当自定义结构数据类型为指针类型时使用_pst 后缀；

【规则 1-9】含义标识命名规则，变量命名使用名词性词组，函数命名使用动词性词组。

例如：

No	变量名	目标词	动词(的过去分词)	状语	目的地	含义
1	DataGotFromSD	Data	Got	From	SD	从 SD 中取得的数据
2	DataDeletedFromSD	Data	Deleted	From	SD	从 SD 中删除的数据

变量含义标识符构成：目标词 ＋ 动词(的过去分词)+ [状语] + [目的地]；

No	变量名	动词(一般现时)	目标词	状语	目的地	含义
1	GetDataFromSD	Get	Data	From	SD	从 SD 中取得数据
2	DeleteDataFromSD	Delete	Data	From	SD	从 SD 中删除数据

函数含义标识符构成：动词(一般现时)+目标词+[状语]+[目的地]；

【规则 1-10】程序中不得出现仅靠大小写区分的相似的标识符。

例如：int x, X; 变量 x 与 X 容易混淆

void foo(int x); 函数 foo 与 FOO 容易混淆

void FOO(float x);

这里还有一个要特别注意的就是 1（数字 1）和 l（小写字母 l）之间，0（数字 0）和 o（小写字母 o）之间的区别。这两对真是很难区分的，我曾经的一个同事就被这个问题折腾了一次。

【规则 1-11】一个函数名禁止被用于其它之处。

例如：

```
#include "c_standards.h"


void foo(int p_1)
{
    int x = p_1;
}


void static_p(void)
{
    int foo = 1u;
}
```

【规则 1-12】所有宏定义、枚举常数、只读变量全用大写字母命名，用下划线分割单词。

例如：

```
const   int   MAX_LENGTH = 100; //这不是常量，而是一个只读变量，具体请往后看
#define FILE_PATH   “/usr/tmp”
```

【规则 1-13】考虑到习惯性问题，局部变量中可采用通用的命名方式，仅限于 n、i、j 等作为循环变量使用。

一定不要写出如下这样的代码：

```
int   p;
char   i;
int    c;
char * a;
```

一般来说习惯上用 n,m,i,j,k 等表示 int 类型的变量；c，ch 等表示字符类型变量；a 等表示数组；p 等表示指针。当然这仅仅是一般习惯，除了 i,j,k 等可以用来表示循环变量外，别的字符变量名尽量不要使用。

【规则 1-14】定义变量的同时千万千万别忘了初始化。定义变量时编译器并不一定清空了这块内存，它的值可能是无效的数据。

这个问题在内存管理那章有非常详细的讨论，请参看。

【规则 1-15】不同类型数据之间的运算要注意精度扩展问题，一般低精度数据将向高精度数据扩展。

1.5，最冤枉的关键字----sizeof

1.5.1，常年被人误认为函数

sizeof 是关键字不是函数，其实就算不知道它是否为 32 个关键字之一时，我们也可以借助编译器确定它的身份。看下面的例子：

int i=0；

A),sizeof(int)； B)，sizeof(i)； C)，sizeof int； D)，sizeof i；

毫无疑问，32 位系统下 A)，B)的值为 4。那 C)的呢？D)的呢？

在 32 位系统下，通过 Visual C++6.0 或任意一编译器调试，我们发现 D)的结果也为 4。咦？sizeof 后面的括号呢？没有括号居然也行，那想想，函数名后面没有括号行吗？由此轻易得出 sizeof 绝非函数。

好，再看 C)。编译器怎么怎么提示出错呢？不是说 sizeof 是个关键字，其后面的括号可以没有么？那你想想 sizeof int 表示什么啊？int 前面加一个关键字？类型扩展？明显不正确，我们可以在 int 前加 unsigned，const 等关键字但不能加 sizeof。好，记住：sizeof 在计算**变量**所占空间大小时，括号可以省略，而计算**类型(模子)**大小时不能省略。一般情况下，咱也别偷这个懒，乖乖的写上括号，继续装作一个“函数”，做一个“披着函数皮的关键字”。做我的关键字，让人家认为是函数去吧。

1.5.2，sizeof（int）*p 表示什么意思？

sizeof（int）*p 表示什么意思？

留几个问题(讲解指针与数组时会详细讲解)，32 位系统下：

int *p = NULL;

sizeof(p)的值是多少？

sizeof(*p)呢？

```
int a[100];
sizeof (a)  的值是多少？
sizeof(a[100])呢？//请尤其注意本例。
sizeof(&a)呢？
sizeof(&a[0])呢？

int b[100];
void fun(int b[100])
{
    sizeof(b);// sizeof (b)  的值是多少？

}
```

1.4，signed、unsigned 关键字

我们知道计算机底层只认识 0、1.任何数据到了底层都会变计算转换成 0、1.那负数怎么存储呢？肯定这个“-”号是无法存入内存的，怎么办？很好办，做个标记。把基本数据类型的最高位腾出来，用来存符号，同时约定如下：最高位如果是 1，表明这个数是负数，其值为除最高位以外的剩余位的值添上这个“-”号；如果最高位是 0，表明这个数是正数，其值为除最高位以外的剩余位的值。

这样的话，一个 32 位的 signed　int 类型整数其值表示法范围为：-2^{31}～$2^{31}-1$；8 位的 char 类型数其值表示的范围为-2^{7}～$2^{7}-1$。一个 32 位的 unsigned　int 类型整数其值表示法范围为：0～ $2^{32}-1$；8 位的 char 类型数其值表示的范围为 0～$2^{8}-1$。同样我们的 signed 关键字也很宽恒大量，你也可以完全当它不存在，编译器缺省默认情况下数据为 signed 类型的。

上面的解释很容易理解，下面就考虑一下这个问题：

```
int main()
{
   char a[1000];
   int i;
   for(i=0; i<1000; i++)
   {
       a[i] = -1-i;
   }
   printf("%d",strlen(a));
   return 0;
```

}

此题看上去真的很简单，但是却鲜有人答对。答案是 255。别惊讶，我们先分析分析。

for 循环内，当 i 的值为 0 时，a[0]的值为-1。关键就是-1 在内存里面如何存储。

我们知道在计算机系统中，数值一律用补码来表示（存储）。主要原因是使用补码，可以将符号位和其它位统一处理；同时，减法也可按加法来处理。另外，两个用补码表示的数相加时，如果最高位（符号位）有进位，则进位被舍弃。正数的补码与其原码一致；负数的补码：符号位为 1，其余位为该数绝对值的原码按位取反，然后整个数加 1。

按照负数补码的规则，可以知道-1 的补码为 0xff，-2 的补码为 0xfe……当 i 的值为 127 时，a[127]的值为-128，而-128 是 char 类型数据能表示的最小的负数。当 i 继续增加，a[128]的值肯定不能是-129。因为这时候发生了溢出，-129 需要 9 位才能存储下来，而 char 类型数据只有 8 位，所以最高位被丢弃。剩下的 8 位是原来 9 位补码的低 8 位的值，即 0x7f。当 i 继续增加到 255 的时候，-256 的补码的低 8 位为 0。然后当 i 增加到 256 时，-257 的补码的低 8 位全为 1，即低八位的补码为 0xff，如此又开始一轮新的循环……

按照上面的分析，a[0]到 a[254]里面的值都不为 0，而 a[255]的值为 0。strlen 函数是计算字符串长度的，并不包含字符串最后的‘\0’。而判断一个字符串是否结束的标志就是看是否遇到‘\0’。如果遇到‘\0’，则认为本字符串结束。

分析到这里，strlen(a)的值为 255 应该完全能理解了。这个问题的关键就是要明白 char 类型默认情况下是有符号的，其表示的值的范围为[-128,127]，超出这个范围的值会产生溢出。另外还要清楚的就是负数的补码怎么表示。弄明白了这两点，这个问题其实就很简单了。

留三个问题：

1），按照我们上面的解释，那-0 和+0 在内存里面分别怎么存储？

2），int i = -20;

```
unsigned    j = 10;
```

i+j 的值为多少？为什么？

3）， 下面的代码有什么问题？

```
unsigned i ;
for (i=9;i>=0;i--)
{
    printf("%u\n",i);
}
```

1.6，if、else 组合

if 语句很简单吧。嗯，的确很简单。那我们就简单的看下面几个简单的问题：

1.6.1，bool 变量与“零值”进行比较

bool 变量与“零值”进行比较的 if 语句怎么写？

bool　bTestFlag = FALSE;//想想为什么一般初始化为 FALSE 比较好？

A), if(bTestFlag == 0);　　　　if(bTestFlag == 1);

B), if(bTestFlag == TRUE);　　if(bTestFlag == FLASE);

C), if(bTestFlag);　　　　　　if(!bTestFlag);

哪一组或是那些组正确呢？我们来分析分析：

A)写法：bTestFlag 是什么？整型变量？如果要不是这个名字遵照了前面的命名规范，肯怕很容易让人误会成整型变量。所以这种写法不好。

B)写法：FLASE 的值大家都知道，在编译器里被定义为 0；但 TRUE 的值呢？都是 1 吗？很不幸，不都是 1。Visual C++定义为 1，而它的同胞兄弟 Visual Basic 就把 TRUE 定义为-1.那很显然，这种写法也不好。

大家都知道 if 语句是靠其后面的括号里的表达式的值来进行分支跳转的。表达式如果为真，则执行 if 语句后面紧跟的代码；否则不执行。那显然，本组的写法很好，既不会引起误会，也不会由于 TRUE 或 FLASE 的不同定义值而出错。记住：以后写代码就得这样写。

1.6.2，float 变量与“零值”进行比较

float 变量与“零值”进行比较的 if 语句怎么写？

float fTestVal = 0.0;

A), if(fTestVal == 0.0);　　if(fTestVal != 0.0);

B), if((fTestVal >= -EPSINON) && 　(fTestVal　<= EPSINON)); //EPSINON 为定义好的精度。

哪一组或是那些组正确呢？我们来分析分析：

float 和 double 类型的数据都是有精度限制的，这样直接拿来与 0.0 比，能正确吗？明显不能，看例子：　π 的值四舍五入精确到小数点后 10 位为：3.1415926536，你拿它减去 0.00000000001 然后再四舍五入得到的结果是多少?你能说前后两个值一样吗?

EPSINON 为定义好的精度，如果一个数落在[0.0-EPSINON,0.0+EPSINON] 这个闭区间内，我们认为在某个精度内它的值与零值相等；否则不相等。扩展一下，把 0.0 替换为你想比较的任何一个浮点数，那我们就可以比较任意两个浮点数的大小了，当然是在某个精度内。

同样的也不要在很大的浮点数和很小的浮点数之间进行运算，比如：
10000000000.00 + 0.00000000001
这样计算后的结果可能会让你大吃一惊。

1.6.3，指针变量与“零值”进行比较

指针变量与“零值”进行比较的 if 语句怎么写？

int * p = NULL;//定义指针一定要同时初始化，指针与数组那章会详细讲解。

A), if(p == 0);   if(p != 0);

B), if(p);   if(!p);

C) , if(NULL == p);   if(NULL != p);

哪一组或是那些组正确呢？我们来分析分析：

A)写法：p 是整型变量？容易引起误会，不好。尽管 NULL 的值和 0 一样，但意义不同。

B)写法：p 是 bool 型变量？容易引起误会，不好。

C)写法：这个写法才是正确的，但样子比较古怪。为什么要这么写呢？是怕漏写一个“=”号:if(p = NULL)，这个表达式编译器当然会认为是正确的，但却不是你要表达的意思。所以，非常推荐这种写法。

1.6.4，else 到底与哪个 if 配对呢？

else 常常与 if 语句配对，但要注意书写规范，看下面例子：

```
if（0 == x）
if（0 == y）  error（）;
else{
    //program code
}
```

这个 else 到底与谁匹配呢？让人迷糊，尤其是初学者。还好，C 语言有这样的规定：else 始终与同一括号内最近的未匹配的 if 语句结合。虽然老手可以区分出来，但这样的代码谁都会头疼的，任何时候都别偷这种懒。关于程序中的分界符‘{’ 和 ‘}’，建议如下：

【建议 1-16】程序中的分界符‘{’ 和 ‘}’对齐风格如下：

注意下表中代码的缩进一般为 4 个字符，但不要使用 Tab 键，因为不同的编辑器 Tab 键定义的空格数量不一样，别的编辑器打开 Tab 键缩进的代码可能会一片混乱。

提倡的的风格	不提倡的风格

void Function(int x)
{
    //program code
}
	void Function(int x){
	//program code
	}

if (condition)
{
    //program code
}
	if (condition){
	//program code
	}else{
	//program code

else
{
    //program code
}
	}
	或：
	if (condition)
	//program code
	else
	//program code
	或：
	if (width < height) dosomething();

for (initialization; condition; update)
{
    //program code
}
	for (initialization;condition; update){
	//program code
	}

while (condition)
{
    //program code
}
	while (condition){
	//program code
	}

do
{
    //program code
}
while (condition);
	do{
	//program code
	}while (condition);

1.6.5，if 语句后面的分号

关于 if-else 语句还有一个容易出错的地方就是与空语句的连用。看下面的例子：

```
if(NULL != p) ;
    fun();
```

这里的 fun()函数并不是在 NULL != p 的时候被调用，而是任何时候都会被调用。问题就出在 if 语句后面的分号上。在 C 语言中，分号预示着一条语句的结尾，但是并不是每条 C 语言语句都需要分号作为结束标志。if 语句的后面并不需要分号，但如果你不小心写了个分号，编译器并不会提示出错。因为编译器会把这个分号解析成一条空语句。也就是上面的代码实际等效于：

```
if(NULL != p)
{
```

```
        ;
    }
    fun();
```

这是初学者很容易犯的错误，往往不小心多写了个分号，导致结果与预想的相差很远。所以建议在真正需要用空语句时写成这样：

```
    NULL;
```

而不是单用一个分号。这就好比汇编语言里面的空指令，比如 ARM 指令中的 NOP 指令。这样做可以明显的区分真正必须的空语句和不小心多写的分号。

1.6.6，使用 if 语句的其他注意事项

【规则 1-17】先处理正常情况，再处理异常情况。

在编写代码是，要使得正常情况的执行代码清晰，确认那些不常发生的异常情况处理代码不会遮掩正常的执行路径。这样对于代码的可读性和性能都很重要。因为，if 语句总是需要做判断，而正常情况一般比异常情况发生的概率更大（否则就应该把异常正常调过来了），如果把执行概率更大的代码放到后面，也就意味着 if 语句将进行多次无谓的比较。另外，非常重要的一点是，把正常情况的处理放在 if 后面，而不要放在 else 后面。当然这也符合把正常情况的处理放在前面的要求。

【规则 1-18】确保 if 和 else 子句没有弄反。

这一点初学者也容易弄错，往往把本应该放在 if 语句后面的代码和本应该放在 else 语句后面的代码弄反了。

1.7，switch、case 组合

既然有了 if、else 组合为什么还需要 switch、case 组合呢？

1.7.1，不要拿青龙偃月刀去削苹果

那你既然有了菜刀为什么还需要水果刀呢？你总不能扛着云长的青龙偃月刀（又名冷艳锯）去削苹果吧。如果你真能做到，关二爷也会佩服你的。^_^。

if、else 一般表示两个分支或是嵌套表示少量的分支，但如果分支很多的话……还是用 switch、case 组合吧。其基本格式为:

```
switch(variable)
{
    case Value1:
        //program code
```

```
            break;
      case Value2:
            //program code
            break;
      case Value3:
            //program code
            break;
        …
    default:
            break;

}
```

很简单，但有两个规则：

【规则 1-19】每个 case 语句的结尾绝对不要忘了加 break，否则将导致多个分支重叠（除非有意使多个分支重叠）。

【规则 1-20】最后必须使用 default 分支。即使程序真的不需要 default 处理，也应该保留语句：

```
default :
      break;
```

这样做并非画蛇添足，可以避免让人误以为你忘了 default 处理。

1.7.2，case 关键字后面的值有什么要求吗？

好，再问问：真的就这么简单吗？看看下面的问题：

Value1 的值为 0.1 行吗？-0.1 呢？-1 呢？0.1+0.9 呢？ 1+2 呢？3/2 呢？ ‘A’呢？“A”呢？变量 i（假设 i 已经被初始化）呢？NULL 呢？等等。这些情形希望你亲自上机调试一下，看看到底哪些行，哪些不行。

记住：case 后面只能是整型或字符型的常量或常量表达式（想想字符型数据在内存里是怎么存的）。

1.7.3，case 语句的排列顺序

似乎从来没有人考虑过这个问题，也有很多人认为 case 语句的顺序无所谓。但事实却不是如此。如果 case 语句很少，你也许可以忽略这点，但是如果 case 语句非常多，那就不得不好好考虑这个问题了。比如你写的是某个驱动程序，也许会经常遇到几十个 case 语句的情况。一般来说，我们可以遵循下面的规则：

【规则 1-21】按字母或数字顺序排列各条 case 语句。

如果所有的 case 语句没有明显的重要性差别，那就按 A-B-C 或 1-2-3 等顺序排列 case 语句。这样做的话，你可以很容易的找到某条 case 语句。比如：

```
switch(variable)
{
    case  A:
        //program code
        break;
    case  B:
        //program code
        break;
    case  C:
        //program code
        break;
     …
   default:
        break;

}
```

【规则 1-22】把正常情况放在前面，而把异常情况放在后面。

如果有多个正常情况和异常情况，把正常情况放在前面，并做好注释；把异常情况放在后面，同样要做注释。比如：

```
switch(variable)
{
   //////////////////////////////////////////////////////////////////////////////////
   //正常情况开始
    case  A:
        //program code
        break;
    case  B:
        //program code
        break;
   //正常情况结束
   //////////////////////////////////////////////////////////////////////////////////
```

```
        //异常情况开始
            case   -1:
                //program code
                break;
        //异常情况结束
        //////////////////////////////////////////////////////////////////////////////////
            …
         default:
                break;

    }
```

【规则 1-23】按执行频率排列 case 语句

把最常执行的情况放在前面，而把最不常执行的情况放在后面。最常执行的代码可能也是调试的时候要单步执行的最多的代码。如果放在后面的话，找起来可能会比较困难，而放在前面的话，可以很快的找到。

1.7.4，使用 case 语句的其他注意事项

【规则 1-24】简化每种情况对应的操作。

使得与每种情况相关的代码尽可能的精炼。case 语句后面的代码越精炼，case 语句的结果就会越清晰。你想想，如果 case 语句后面的代码整个屏幕都放不下，这样的代码谁也难看得很清晰吧。如果某个 case 语句确实需要这么多的代码来执行某个操作，那可以把这些操作写成一个或几个子程序，然后在 case 语句后面调用这些子程序就 ok 了。一般来说 case 语句后面的代码尽量不要超过 20 行。

【规则 1-25】不要为了使用 case 语句而刻意制造一个变量。

case 语句应该用于处理简单的，容易分类的数据。如果你的数据并不简单，那可能使用if-else if 的组合更好一些。为了使用 case 而刻意构造出来的变量很容易把人搞糊涂，应该避免这种变量。比如：

```
char action = a[0];
switch (action)
{
    case ‘c’:
        fun1（）;
        break;
    case ‘d’:
        …
        break;
    default:
```

```
            break;

    }
```

这里控制 case 语句的变量是 action。而 action 的值是取字符数组 a 的一个字符。但是这种方式可能带来一些隐含的错误。一般而言，当你为了使用 case 语句而刻意去造出一个变量时，真正的数据可能不会按照你所希望的方式映射到 case 语句里。在这个例子中，如果用户输入字符数组 a 里面存的是“const”这个字符串，那么 case 语句会匹配到第一个 case 上，并调用 fun1（）函数。然而如果这个数组里存的是别的以字符 c 开头的任何字符串（比如：“col”，“can”），case 分支同样会匹配到第一个 case 上。但是这也许并不是你想要的结果，这个隐含的错误往往使人抓狂。如果这样的话还不如使用 if-else if 组合。比如：

```
if（0 == strcmp(“const”，a)）
{
    fun1();
}
else if
{
   …
}
```

【规则 1-26】把 default 子句只用于检查真正的默认情况。

有时候，你只剩下了最后一种情况需要处理，于是就决定把这种情况用 default 子句来处理。这样也许会让你偷懒少敲几个字符，但是这却很不明智。这样将失去 case 语句的标号所提供的自说明功能，而且也丧失了使用 default 子句处理错误情况的能力。所以，奉劝你不要偷懒，老老实实的把每一种情况都用 case 语句来完成，而把真正的默认情况的处理交给 default 子句。

1.8，do、while、for 关键字

C 语言中循环语句有三种：while 循环、do-while 循环、for 循环。

while 循环：先判断 while 后面括号里的值，如果为真则执行其后面的代码；否则不执行。while（1）表示死循环。死循环有没有用呢？看下面例子：

比如你开发一个系统要日夜不停的运行，但是只有操作员输入某个特定的字符‘#’才可以停下来。

```
while（1）
{
    if(‘#’== GetInputChar())
    {
        break;
```

```
    }
}
```

1.8.1，break 与 continue 的区别

break 关键字很重要，表示终止**本层**循环。现在这个例子只有一层循环，当代码执行到 break 时，循环便终止。

如果把 break 换成 continue 会是什么样子呢？continue 表示终止**本次（本轮）**循环。当代码执行到 continue 时，本轮循环终止，进入下一轮循环。

while（1）也有写成 while(true) 或者 while(1==1) 或者 while((bool) 1)等形式的，效果一样。

do-while 循环：先执行 do 后面的代码，然后再判断 while 后面括号里的值，如果为真，循环开始；否则，循环不开始。其用法与 while 循环没有区别，但相对较少用。

for 循环：for 循环可以很容易的控制循环次数，多用于事先知道循环次数的情况下。

留一个问题：在 switch case 语句中能否使用 continue 关键字？为什么？

1.8.2，循环语句的注意点

【建议 1-27】在多重循环中，如果有可能，应当将最长的循环放在最内层，最短的循环放在最外层，以减少 CPU 跨切循环层的次数。

例如：

长循环在最内层，效率高	长循环在最外层，效率低

for (col=0; col<5; col++)
{
    for (row=0; row<100; row++)
    {
        sum = sum + a[row][col];
    }
}
	for (row=0; row<100; row++)
	{
	    for (col=0; col<5; col++)
	    {
	        sum = sum + a[row][col];
	    }
	}

【建议 1-28】建议 for 语句的循环控制变量的取值采用“半开半闭区间”写法。

半开半闭区间写法和闭区间写法虽然功能是相同，但相比之下，半开半闭区间写法写法更加直观。

半开半闭区间写法	闭区间写法
for (n = 0; n < 10; n++)	for (n = 0; n <= 9; n++)

{
 …
 }
	{
	 …
	 }

【规则 1-29】不能在 for 循环体内修改循环变量，防止循环失控。

```
for (n = 0; n < 10; n++)
{
    …
    n = 8;//不可，很可能违背了你的原意
    …
}
```

【规则 1-30】循环要尽可能的短，要使代码清晰，一目了然。

如果你写的一个循环的代码超过一显示屏，那会让读代码的人发狂的。解决的办法由两个：第一，重新设计这个循环，确认是否这些操作都必须放在这个循环里；第二，将这些代码改写成一个子函数，循环中只调用这个子函数即可。一般来说循环内的代码不要超过 20 行。

【规则 1-31】把循环嵌套控制在 3 层以内。

国外有研究数据表明，当循环嵌套超过 3 层，程序员对循环的理解能力会极大的降低。如果你的循环嵌套超过 3 层，建议你重新设计循环或是将循环内的代码改写成一个字函数。

1.9，goto 关键字

一般来说，编码的水平与 goto 语句使用的次数成反比。有的人主张慎用但不禁用 goto 语句，但我主张禁用。关于 goto 语句的更多讨论可以参看 Steve McConnell 的名著《Code Complete. Second Edition》。

【规则 1-32】禁用 goto 语句。

自从提倡结构化设计以来，goto 就成了有争议的语句。首先，由于 goto 语句可以灵活跳转，如果不加限制，它的确会破坏结构化设计风格；其次，goto 语句经常带来错误或隐患。它可能跳过了变量的初始化、重要的计算等语句，例如：

```
struct student *p = NULL;
…
goto state;
p = (struct student *)malloc(…); //被 goto 跳过,没有初始化
```

```
…
state:
//使用 p 指向的内存里的值的代码
…
```

如果编译器不能发觉此类错误，每用一次 goto 语句都可能留下隐患。

1.10，void 关键字

void 有什么好讲的呢？如果你认为没有，那就没有；但如果你认为有，那就真的有。有点像“色即是空，空即是色”。

1.10.1，void a?

void 的字面意思是“空类型”，void *则为“空类型指针”，void *可以指向任何类型的数据。void 几乎只有“注释”和限制程序的作用，因为从来没有人会定义一个 void 变量，看看下面的例子：

```
void a;
```

Visual C++6.0 上，这行语句编译时会出错，提示“illegal use of type 'void'”。不过，即使 void a 的编译不会出错，它也没有任何实际意义。

void 真正发挥的作用在于：

（1） 对函数返回的限定；

（2） 对函数参数的限定。

众所周知，如果指针 p1 和 p2 的类型相同，那么我们可以直接在 p1 和 p2 间互相赋值；如果 p1 和 p2 指向不同的数据类型，则必须使用强制类型转换运算符把赋值运算符右边的指针类型转换为左边指针的类型。

例如：

```
float   *p1;
int   *p2;
p1 = p2;
```

其中 p1 = p2 语句会编译出错，提示“'=' : cannot convert from 'int *' to 'float *'”，必须改为：

```
p1 = (float *)p2;
```

而 void *则不同，任何类型的指针都可以直接赋值给它，无需进行强制类型转换：

```
void   *p1;
int   *p2;
p1 = p2;
```

但这并不意味着，void *也可以无需强制类型转换地赋给其它类型的指针。因为“空类型”可以包容“有类型”，而“有类型”则不能包容“空类型”。比如，我们可以说“男人和女人都是人”，但不能说“人是男人”或者“人是女人”。下面的语句编译出错：

```
void    *p1;
int    *p2;
p2 = p1;
```
提示“'=' : cannot convert from 'void *' to 'int *'”。

1.10.2，void 修饰函数返回值和参数

【规则 1-33】如果函数没有返回值，那么应声明为 void 类型

在 C 语言中，凡不加返回值类型限定的函数，就会被编译器作为返回整型值处理。但是许多程序员却误以为其为 void 类型。例如：

```
add ( int a, int b )
{
    return    a + b;
}


int main(int argc, char* argv[]) //甚至很多人以为 main 函数无返回值
                                //或是为 void 型的
{
    printf ( "2 + 3 = %d", add ( 2, 3) );
}
```
程序运行的结果为输出：　2 + 3 = 5

这说明不加返回值说明的函数的确为 int 函数。

因此，为了避免混乱，我们在编写 C 程序时，对于任何函数都必须一个不漏地指定其类型。如果函数没有返回值，一定要声明为 void 类型。这既是程序良好可读性的需要，也是编程规范性的要求。另外，加上 void 类型声明后，也可以发挥代码的“自注释”作用。所谓的代码的“自注释”即代码能自己注释自己。

【规则 1-34】如果函数无参数,那么应声明其参数为 void

在 C++语言中声明一个这样的函数：

```
int function(void)
{
    return 1;
}
```
则进行下面的调用是不合法的：function(2);

因为在 C++中，函数参数为 void 的意思是这个函数不接受任何参数。

但是在 Turbo C 2.0 中编译：

```
#include "stdio.h"
fun()
{
    return 1;
}
main()
{
    printf("%d",fun(2));
    getchar();
```

}

编译正确且输出 1，这说明，在 C 语言中，可以给无参数的函数传送任意类型的参数，但是在 C++编译器中编译同样的代码则会出错。在 C++中，不能向无参数的函数传送任何参数，出错提示“'fun' : function does not take 1 parameters”。

所以，无论在 C 还是 C++中，若函数不接受任何参数，一定要指明参数为 void。

1.10.3，void 指针

【规则 1-35】千万小心又小心使用 void 指针类型。

按照 ANSI(American National Standards Institute)标准，不能对 void 指针进行算法操作，即下列操作都是不合法的：

```
void * pvoid;
pvoid++; //ANSI：错误
pvoid += 1; //ANSI：错误
```

ANSI 标准之所以这样认定，是因为它坚持：进行算法操作的指针必须是确定知道其指向数据类型大小的。也就是说必须知道内存目的地址的确切值。

例如：

```
int *pint;
pint++; //ANSI：正确
```

但是大名鼎鼎的 GNU(GNU's Not Unix 的递归缩写)则不这么认定，它指定 void *的算法操作与 char *一致。因此下列语句在 GNU 编译器中皆正确：

```
pvoid++; //GNU：正确
pvoid += 1; //GNU：正确
```

在实际的程序设计中，为符合 ANSI 标准，并提高程序的可移植性，我们可以这样编写实现同样功能的代码：

```
void * pvoid;
(char *)pvoid++; //ANSI：正确；GNU：正确
(char *)pvoid += 1; //ANSI：错误；GNU：正确
```

GNU 和 ANSI 还有一些区别，总体而言，GNU 较 ANSI 更“开放”，提供了对更多语法的支持。但是我们在真实设计时，还是应该尽可能地符合 ANSI 标准。

【规则 1-36】如果函数的参数可以是任意类型指针，那么应声明其参数为 void *。

典型的如内存操作函数 memcpy 和 memset 的函数原型分别为：

```
void * memcpy(void *dest, const void *src, size_t len);
void * memset ( void * buffer, int c, size_t num );
```

这样，任何类型的指针都可以传入 memcpy 和 memset 中，这也真实地体现了内存操作函数的意义，因为它操作的对象仅仅是一片内存，而不论这片内存是什么类型。如果 memcpy 和 memset 的参数类型不是 void *，而是 char *，那才叫真的奇怪了！这样的 memcpy 和 memset 明显不是一个“纯粹的，脱离低级趣味的”函数！

下面的代码执行正确：

例子：memset 接受任意类型指针

```
int IntArray_a[100];
memset (IntArray_a, 0, 100*sizeof(int) ); //将 IntArray_a 清 0
```

例子：memcpy 接受任意类型指针

```
int destIntArray_a[100], srcintarray_a[100];
```

```
//将 srcintarray_a 拷贝给 destIntArray_a
memcpy (destIntArray_a, srcintarray_a, 100*sizeof(int) );
```

有趣的是，memcpy 和 memset 函数返回的也是 void *类型，标准库函数的编写者都不是一般人。

1.10.4，void 不能代表一个真实的变量

【规则 1-37】void 不能代表一个真实的变量。

因为定义变量时必须分配内存空间，定义 void 类型变量，编译器到底分配多大的内存呢。

下面代码都企图让 void 代表一个真实的变量，因此都是错误的代码：

```
void a; //错误
function(void a); //错误
```

void 体现了一种抽象，这个世界上的变量都是“有类型”的，譬如一个人不是男人就是女人（人妖不算）。

void 的出现只是为了一种抽象的需要，如果你正确地理解了面向对象中“抽象基类”的概念，也很容易理解 void 数据类型。正如不能给抽象基类定义一个实例，我们也不能定义一个 void（让我们类比的称 void 为“抽象数据类型”）变量。

void 简单吧？到底是“色”还是“空”呢？

1.10，return 关键字

return 用来终止一个函数并返回其后面跟着的值。

return （Val）；//此括号可以省略。但一般不省略，尤其在返回一个表达式的值时。

return 可以返回些什么东西呢？看下面例子：

```
char * Func(void)
{
    char str[30];
    …
    return str;
}
```

str 属于局部变量，位于栈内存中，在 Func 结束的时候被释放，所以返回 str 将导致错误。

【规则 1-38】return 语句不可返回指向“栈内存”的“指针”，因为该内存在函数体结束时被自动销毁。

留个问题：

```
return ；
```

这个语句有问题吗？如果没有问题，那返回的是什么？

1.11，const 关键字也许该被替换为 readolny

const 是 constant 的缩写，是恒定不变的意思，也翻译为常量、常数等。很不幸，正是因为这一点，很多人都认为被 const 修饰的值是常量。这是不精确的，精确的说应该是只读的变量，其值在编译时不能被使用，因为编译器在编译时不知道其存储的内容。或许当初这个关键字应该被替换为 readonly。那么这个关键字有什么用处和意义呢？

const 推出的初始目的，正是为了取代预编译指令，消除它的缺点，同时继承它的优点。我们看看它与 define 宏的区别。（很多人误以为 define 是关键字，在这里我提醒你再回到本章前面看看 32 个关键字里是否有 define）。

1.11.1，const 修饰的只读变量

定义 const 只读变量，具有不可变性。

例如：

```
const int Max=100;

int Array[Max];
```

这里请在 Visual C++6.0 里分别创建.c 文件和.cpp 文件测试一下。你会发现在.c 文件中，编译器会提示出错，而在.cpp 文件中则顺利运行。为什么呢？我们知道定义一个数组必须指定其元素的个数。这也从侧面证实在 C 语言中，const 修饰的 Max 仍然是变量，只不过是只读属性罢了；而在 C++里，扩展了 const 的含义，这里就不讨论了。

注意：const 修饰的只读变量必须在定义的同时初始化，想想为什么？

留一个问题：case 语句后面是否可以是 const 修饰的只读变量呢？请动手测试一下。

1.11.2，节省空间，避免不必要的内存分配，同时提高效率

编译器通常不为普通 const 只读变量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的值，没有了存储与读内存的操作，使得它的效率也很高。

例如：

```
#define M 3          //宏常量

const int N=5;     //此时并未将 N 放入内存中
......
int i=N;             //此时为 N 分配内存，以后不再分配！
int I=M;             //预编译期间进行宏替换，分配内存
int j=N;             //没有内存分配
int J=M;             //再进行宏替换，又一次分配内存！
```

const 定义的只读变量从汇编的角度来看，只是给出了对应的内存地址，而不是象#define 一样给出的是立即数，所以，const 定义的只读变量在程序运行过程中只有一份拷贝（因为它是全局的只读变量，存放在静态区），而#define 定义的宏常量在内存中有若干个拷贝。

#define 宏是在预编译阶段进行替换，而 const 修饰的只读变量是在编译的时候确定其值。

#define 宏没有类型，而 const 修饰的只读变量具有特定的类型。

1.11.3，修饰一般变量

一般常量是指简单类型的只读变量。这种只读变量在定义时，修饰符 const 可以用在类型说明符前，也可以用在类型说明符后。例如：

int const i=2; 或 const int i=2;

1.11.4，修饰数组

定义或说明一个只读数组可采用如下格式：

int const a[5]={1, 2, 3, 4, 5};或

const int a[5]={1, 2, 3, 4, 5};

1.11.5，修饰指针

const int *p; // p 可变，p 指向的对象不可变

int const *p; // p 可变，p 指向的对象不可变

int *const p; // p 不可变，p 指向的对象可变

const int *const p; //指针 p 和 p 指向的对象都不可变

在平时的授课中发现学生很难记住这几种情况。这里给出一个记忆和理解的方法：

先忽略类型名（编译器解析的时候也是忽略类型名），我们看 const 离哪个近。“近水楼台先得月”，离谁近就修饰谁。

const ~~int~~ *p; //const 修饰*p,p 是指针，*p 是指针指向的对象，不可变

~~int~~ const *p; //const 修饰*p,p 是指针，*p 是指针指向的对象，不可变

~~int~~ *const p; //const 修饰 p，p 不可变，p 指向的对象可变

const ~~int~~ *const p; //前一个 const 修饰*p,后一个 const 修饰 p，指针 p 和 p 指向的对象都不可变

1.11.6，修饰函数的参数

const 修饰符也可以修饰函数的参数，当不希望这个参数值被函数体内意外改变时使用。例如：

void Fun(const int i);

告诉编译器 i 在函数体中的不能改变，从而防止了使用者的一些无意的或错误的修改。

1.11.7，修饰函数的返回值

const 修饰符也可以修饰函数的返回值，返回值不可被改变。例如：

const int Fun (void);

在另一连接文件中引用 const 只读变量：

extern const int i; //正确的声明

extern const int j=10; //错误！只读变量的值不能改变。

注意这里是声明不是定义，关于声明和定义的区别，请看本章开始处。

讲了这么多讲完了吗？远没有。在 C++里，对 const 做了进一步的扩展，还有很多知识未能

讲完。有兴趣的话，不妨查找相关资料研究研究。

1.12，最易变的关键字----volatile

volatile 是易变的、不稳定的意思。很多人根本就没见过这个关键字，不知道它的存在。也有很多程序员知道它的存在，但从来没用过它。我对它有种“杨家有女初长成,养在深闺人未识” 的感觉。

volatile 关键字和 const 一样是一种类型修饰符，用它修饰的变量表示可以被某些编译器未知的因素更改，比如操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。

先看看下面的例子：

```
int i=10;
int j = i；//(1)语句
int k = i；//(2)语句
```

这时候编译器对代码进行优化，因为在（1）、（2）两条语句中，i 没有被用作左值。这时候编译器认为 i 的值没有发生改变，所以在（1）语句时从内存中取出 i 的值赋给 j 之后，这个值并没有被丢掉，而是在（2）语句时继续用这个值给 k 赋值。编译器不会生成出汇编代码重新从内存里取 i 的值，这样提高了效率。但要注意：（1）、（2）语句之间 i 没有被用作左值才行。

再看另一个例子：

```
volatile    int i=10;
int j = i；//(3)语句
int k = i；//(4)语句
```

volatile 关键字告诉编译器 i 是随时可能发生变化的，每次使用它的时候必须从内存中取出 i 的值，因而编译器生成的汇编代码会重新从 i 的地址处读取数据放在 k 中。

这样看来，如果 i 是一个寄存器变量或者表示一个端口数据或者是多个线程的共享数据，就容易出错，所以说 volatile 可以保证对特殊地址的稳定访问。

但是注意：在 VC++6.0 中，一般 Debug 模式没有进行代码优化，所以这个关键字的作用有可能看不出来。你可以同时生成 Debug 版和 Release 版的程序做个测试。

留一个问题：const volatile int i=10；这行代码有没有问题？如果没有，那 i 到底是什么属性？

1.13，最会带帽子的关键字----extern

extern，外面的、外来的意思。那它有什么作用呢？举个例子：假设你在大街上看到

一个黑皮肤绿眼睛红头发的美女（外星人？）或者帅哥。你的第一反应就是这人不是国产的。extern 就相当于他们的这些区别于中国人的特性。extern 可以置于变量或者函数前，以标示变量或者函数的定义在别的文件中，下面的代码用到的这些变量或函数是外来的，不是本文件定义的，提示编译器遇到此变量和函数时在其他模块中寻找其定义。就好比在本文件中给这些外来的变量或函数带了顶帽子，告诉本文件中所有代码，这些家伙不是土著。那你想想 extern 修饰的变量或函数是定义还是声明？

看列子：

A.c 文件中定义：	B.c 文件中用 extern 修饰：
int i = 10；	extern int i；//写成 i = 10；行吗？
void fun（void）	extern void fun（void）；//两个 void 可否省略？
{	
//code	
}	

C.h 文件中定义：	D.c 文件中用 extern 修饰：
int j = 1;	extern double j；//这样行吗？为什么？
int k = 2;	j = 3.0；//这样行吗？为什么？

至于 extern“C”的用法，一般认为属于 C++的范畴，这里就先不讨论。当然关于extern 的讨论还远没有结束，在指针与数组那一章，你还会和它亲密接触的。

1.14，struct 关键字

struct 是个神奇的关键字，它将一些相关联的数据打包成一个整体，方便使用。

在网络协议、通信控制、嵌入式系统、驱动开发等地方，我们经常要传送的不是简单的字节流（char 型数组），而是多种数据组合起来的一个整体，其表现形式是一个结构体。经验不足的开发人员往往将所有需要传送的内容依顺序保存在 char 型数组中，通过指针偏移的方法传送网络报文等信息。这样做编程复杂，易出错，而且一旦控制方式及通信协议有所变化，程序就要进行非常细致的修改，非常容易出错。这个时候只需要一个结构体就能搞定。平时我们要求函数的参数尽量不多于 4 个，如果函数的参数多于 4 个使用起来非常容易出错(包括每个参数的意义和顺序都容易弄错)，效率也会降低(与具体 CPU 有关，ARM 芯片对于超过 4 个参数的处理就有讲究，具体请参考相关资料)。这个时候，可以用结构体压缩参数个数。

1.14.1，空结构体多大？

结构体所占的内存大小是其成员所占内存之和（关于结构体的内存对齐，请参考预处理那章)。这点很容易理解，但是下面的这种情况呢？

```
struct student
{
}stu;
```

sizeof(stu)的值是多少呢？在 Visual C++ 6.0 上测试一下。

很遗憾，不是 0，而是 1。为什么呢？你想想，如果我们把 struct student 看成一个模子的话，你能造出一个没有任何容积的模子吗？显然不行。编译器也是如此认为。编译器认为任何一种数据类型都有其大小，用它来定义一个变量能够分配确定大小的空间。既然如此，编译器就理所当然的认为任何一个结构体都是有大小的，哪怕这个结构体为空。那万一结构体真的为空，它的大小为什么值比较合适呢？假设结构体内只有一个 char 型的数据成员，那其大小为 1byte（这里先不考虑内存对齐的情况）.也就是说非空结构体类型数据最少需要占一个字节的空间，而空结构体类型数据总不能比最小的非空结构体类型数据所占的空间大吧。这就麻烦了，空结构体的大小既不能为 0，也不能大于 1，怎么办？定义为 0.5 个 byte？但是内存地址的最小单位是 1 个 byte，0.5 个 byte 怎么处理？解决这个问题的最好办法就是折中，编译器理所当然的认为你构造一个结构体数据类型是用来打包一些数据成员的，而最小的数据成员需要 1 个 byte，编译器为每个结构体类型数据至少预留 1 个 byte 的空间。所以，空结构体的大小就定位 1 个 byte。

1.14.2，柔性数组

也许你从来没有听说过柔性数组（flexible array）这个概念，但是它确实是存在的。

C99 中，结构中的最后一个元素允许是未知大小的数组，这就叫做柔性数组成员，但结构中的柔性数组成员前面必须至少一个其他成员。柔性数组成员允许结构中包含一个大小可变的数组。sizeof 返回的这种结构大小不包括柔性数组的内存。包含柔性数组成员的结构用 malloc ()函数进行内存的动态分配，并且分配的内存应该大于结构的大小，以适应柔性数组的预期大小。

柔性数组到底如何使用呢？看下面例子：

```
typedef struct st_type
{
    int i;
    int a[0];
}type_a;
```

有些编译器会报错无法编译可以改成：

```
typedef struct st_type
{
    int i;
    int a[];
}type_a;
```

这样我们就可以定义一个可变长的结构体，用 sizeof(type_a)得到的只有 4，就是 sizeof(i)=sizeof(int)。那个 0 个元素的数组没有占用空间，而后我们可以进行变长操作了。通过如下表达式给结构体分配内存：

```
type_a *p = (type_a*)malloc(sizeof(type_a)+100*sizeof(int));
```

这样我们为结构体指针 p 分配了一块内存。用 p->item[n]就能简单地访问可变长元素。但是这时候我们再用 sizeof（*p）测试结构体的大小，发现仍然为 4。是不是很诡异？我们不是给这个数组分配了空间么？

别急，先回忆一下我们前面讲过的“模子”。在定义这个结构体的时候，模子的大小就已经确定不包含柔性数组的内存大小。柔性数组只是编外人员，不占结构体的编制。只是说在使用柔性数组时需要把它当作结构体的一个成员，仅此而已。再说白点，柔性数组其实与结构体没什么关系，只是“挂羊头卖狗肉”而已，算不得结构体的正式成员。

需要说明的是：C89 不支持这种东西，C99 把它作为一种特例加入了标准。但是，C99 所支持的是 incomplete type，而不是 zero array，形同 int item[0];这种形式是非法的，C99 支持的形式是形同 int item[];只不过有些编译器把 int item[0];作为非标准扩展来支持，而且在 C99 发布之前已经有了这种非标准扩展了，C99 发布之后，有些编译器把两者合而为一了。

当然，上面既然用 malloc 函数分配了内存，肯定就需要用 free 函数来释放内存：

```
free(p);
```

经过上面的讲解，相信你已经掌握了这个看起来似乎很神秘的东西。不过实在要是没掌握也无所谓，这个东西实在很少用。

1.14.3，struct 与 class 的区别

在 C++里 struct 关键字与 class 关键字一般可以通用，只有一个很小的区别。struct 的成员默认情况下属性是 public 的，而 class 成员却是 private 的。很多人觉得不好记，其实很容易。你平时用结构体时用 public 修饰它的成员了吗？既然 struct 关键字与 class 关键字可以通用，你也不要认为结构体内不能放函数了。

当然，关于结构体的讨论远没有结束，在指针与数组那一章，你还会要和它打交道的。

1.15，union 关键字

union 关键字的用法与 struct 的用法非常类似。

union 维护足够的空间来置放多个数据成员中的“一种”，而不是为每一个数据成员配置空间，在 union 中所有的数据成员共用一个空间，同一时间只能储存其中一个数据成员，所有的数据成员具有相同的起始地址。例子如下：

```
union StateMachine
{
    char character;
    int number;
    char *str;
    double exp;
};
```

一个 union 只配置一个足够大的空间以来容纳最大长度的数据成员，以上例而言，最大长度是 double 型态，所以 StateMachine 的空间大小就是 double 数据类型的大小。

在 C++里，union 的成员默认属性页为 public。union 主要用来压缩空间。如果一些数据

不可能在同一时间同时被用到，则可以使用 union。

1.15.1，大小端模式对 union 类型数据的影响

下面再看一个例子：

```
union
{
    int i;
    char a[2];
}*p, u;

p = &u;
p->a[0] = 0x39;
p->a[1] = 0x38;
```

p.i 的值应该为多少呢？

这里需要考虑存储模式：大端模式和小端模式。

大端模式（Big_endian）：字数据的**高字节**存储在**低地址**中，而字数据的**低字节**则存放在**高地址**中。

小端模式（Little_endian）：字数据的**高字节**存储在**高地址**中，而字数据的**低字节**则存放在**低地址**中。

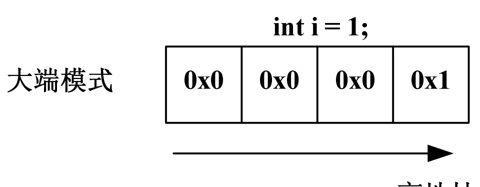
union 型数据所占的空间等于其最大的成员所占的空间。对 union 型的成员的存取都是相对于该联合体基地址的偏移量为 0 处开始，也就是联合体的访问不论对哪个变量的存取都是从 union 的首地址位置开始。如此一解释，上面的问题是否已经有了答案呢？

1.15.2，如何用程序确认当前系统的存储模式？

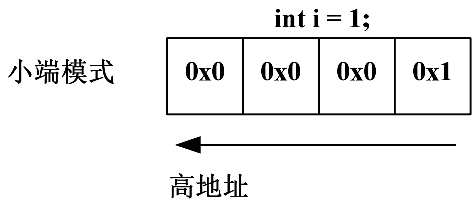
上述问题似乎还比较简单，那来个有技术含量的：请写一个 C 函数，若处理器是 Big_endian 的，则返回 0；若是 Little_endian 的，则返回 1。

先分析一下，按照上面关于大小端模式的定义，假设 int 类型变量 i 被初始化为 1。

以大端模式存储，其内存布局如下图：



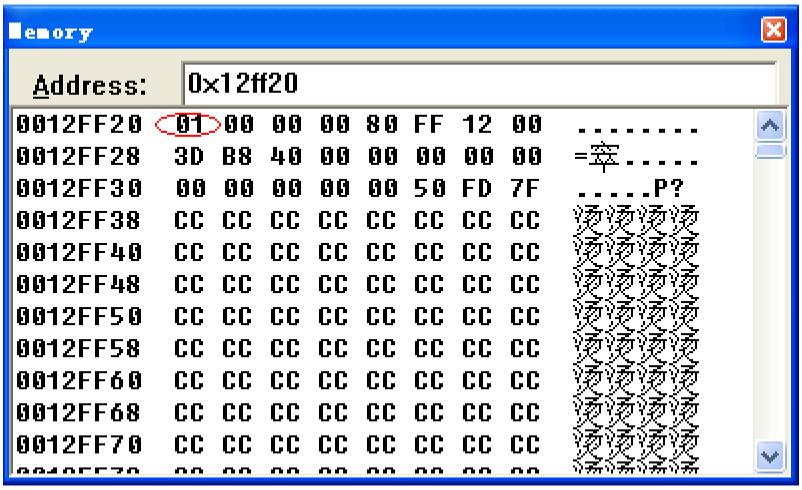
以小端模式存储，其内存布局如下图：



变量 i 占 4 个字节，但只有一个字节的值为 1，另外三个字节的值都为 0。如果取出低地址上的值为 0，毫无疑问，这是大端模式；如果取出低地址上的值为 1，毫无疑问，这是小端模式。既然如此，我们完全可以利用 union 类型数据的特点：所有成员的起始地址一致。到现在，应该知道怎么写了吧？参考答案如下：

```
int checkSystem( )
{
      union check
      {
            int i;
            char ch;
      } c;
     c.i = 1;
     return (c.ch ==1);
}
```

现在你可以用这个函数来测试你当前系统的存储模式了。当然你也可以不用函数而直接去查看内存来确定当前系统的存储模式。如下图：



图中 0x01 的值存在低地址上，说明当前系统为小端模式。

不过要说明的一点是，某些系统可能同时支持这两种存储模式，你可以用硬件跳线或在编译器的选项中设置其存储模式。

留个问题：

在 x86 系统下，输出的值为多少？

```
#include    <stdio.h>
int main()
{
    int a[5]={1,2,3,4,5};
    int *ptr1=(int *)(&a+1);
```

```
    int *ptr2=(int *)((int)a+1);

    printf("%x,%x",ptr1[-1],*ptr2);

    return 0;
}
```

1.16，enum 关键字

很多初学者对枚举(enum)感到迷惑，或者认为没什么用，其实枚举(enum)是个很有用的数据类型。

1.16.1，枚举类型的使用方法

一般的定义方式如下：

```
enum enum_type_name
{
    ENUM_CONST_1,
    ENUM_CONST_2,
    ...
    ENUM_CONST_n
} enum_variable_name;
```

注意：enum_type_name 是自定义的一种数据数据类型名，而 enum_variable_name 为 enum_type_name类型的一个变量，也就是我们平时常说的枚举变量。实际上enum_type_name 类型是对一个变量取值范围的限定，而花括号内是它的取值范围，即 enum_type_name 类型的变量 enum_variable_name 只能取值为花括号内的任何一个值，如果赋给该类型变量的值不在列表中，则会报错或者警告。ENUM_CONST_1、ENUM_CONST_2、...、ENUM_CONST_n，这些成员都是常量，也就是我们平时所说的枚举常量（常量一般用大写）。enum 变量类型还可以给其中的常量符号赋值，如果不赋值则会从被赋初值的那个常量开始依次加 1，如果都没有赋值，它们的值从 0 开始依次递增 1。如分别用一个常数表示不同颜色：

```
enum Color
{
    GREEN = 1,
    RED,
    BLUE,
    GREEN_RED = 10,
    GREEN_BLUE
}ColorVal;
```

其中各常量名代表的数值分别为：

GREEN = 1
RED = 2
BLUE = 3
GREEN_RED = 10
GREEN_BLUE = 11

1.16.2，枚举与#define 宏的区别

下面再看看枚举与#define 宏的区别：

1），#define 宏常量是在预编译阶段进行简单替换。枚举常量则是在编译的时候确定其值。

2），一般在编译器里，可以调试枚举常量，但是不能调试宏常量。

3），枚举可以一次定义大量相关的常量，而#define 宏一次只能定义一个。

留两个问题：

A），枚举能做到事，#define 宏能不能都做到？如果能，那为什么还需要枚举？
B)，sizeof（ColorVal）的值为多少？为什么？

1.17，伟大的缝纫师----typedef 关键字

1.17.1，关于马甲的笑话

有这样一个笑话：一个猎人在河边抓捕一条蛇，蛇逃进了水里。过一会，一个乌龟爬到岸边。猎人一把抓住这个乌龟，大声的说道：小样，别你为你穿了个马甲我就不认识你了！

typedef 关键字是个伟大的缝纫师，擅长做马甲，任何东西穿上这个马甲就立马变样。它可以把狼变成一头羊，也能把羊变成一头狼。甚至还可以把长着翅膀的鸟人变成天使，同样也能把美丽的天使变成鸟人。所以，你千万不要得罪它，一定要掌握它的脾气，不然哪天我把你当鸟人，你可别怪我。^_^。

1.17.2，历史的误会----也许应该是 typerename

很多人认为 typedef 是定义新的数据类型，这可能与这个关键字有关。本来嘛，type 是数据类型的意思；def(ine)是定义的意思，合起来就是定义数据类型啦。不过很遗憾，这种理解是不正确的。也许这个关键字该被替换为“typerename”或是别的词。

typedef 的真正意思是给一个已经存在的**数据类型（注意：是类型**不是变量）取一个别名，而非定义一个新的数据类型。比如：华美绝伦的芍药，就有个别名---“将离”。中国古代男女交往,往往以芍药相赠,表达惜别之情,送芍药就意味着即将分离。所以文人墨客就给芍药取了个意味深长的别名-----“将离”。这个新的名字就表达了那种依依不舍的惜别之情…

这样新的名字与原来的名字相比，就更能表达出想要表达的意思。

在实际项目中，为了方便，可能很多数据类型（尤其是结构体之类的自定义数据类型）需要我们重新取一个适用实际情况的别名。这时候 typedef 就可以帮助我们。例如：

```
typedef   struct student
{
    //code
}Stu_st,*Stu_pst;//命名规则请参考本章前面部分
```

A），struct student　stu1；和 Stu_st stu1；没有区别。

B），struct student　*stu2；和 Stu_pst stu2；和 Stu_st *stu2；没有区别。

这个地方很多初学者迷惑，B）的两个定义为什么相等呢？其实很好理解。我们把“struct student { /*code*/}”看成一个整体，typedef 就是给“struct student {/*code*/}”取了个别名叫“Stu_st”；同时给“struct student { /*code*/} *”取了个别名叫“Stu_pst”。只不过这两个名字同时取而已，好比你给你家小狗取了个别名叫“大黄”，同时你妹妹给小狗带了小帽子，然后给它取了个别名叫“小可爱”。^_^。

好，下面再把 typedef 与 const 放在一起看看：

C),const　Stu_pst　stu3;

D),Stu_pst　const　stu4;

大多数初学者认为 C）里 const 修饰的是 stu3 指向的对象；D）里 const 修饰的是 stu4 这个指针。很遗憾，C）里 const 修饰的并不是 stu3 指向的对象。那 const 这时候到底修饰的是什么呢？我们在讲解 const int i 的时候说过 const 放在类型名“int”前后都行；而 const int *p 与 int * const p 则完全不一样。也就是说，我们看 const 修饰谁都时候完全可以将数据类型名视而不见，当它不存在。反过来再看“const Stu_pst stu3”，Stu_pst 是“struct student { /*code*/} *”的别名，“struct student {/*code*/} *”是一个整体。对于编译器来说，只认为 Stu_pst 是一个类型名，所以在解析的时候很自然的把“Stu_pst”这个数据类型名忽略掉。现在知道 const 到底修饰的是什么了吧？^_^。

1.17.3，typedef 与#define 的区别

噢，上帝！这真要命！别急，要命的还在后面呢。看如下例子：

E），　#define INT32　int

　　unsigned INT32　i = 10；

F），typedef int int32；

　　unsigned int32　j = 10；

其中 F)编译出错，为什么呢？E）不会出错，这很好理解，因为在预编译的时候 INT32 被替换为 int，而 unsigned　int　i = 10；语句是正确的。但是，很可惜，用 typedef 取的别名不支持这种类型扩展。另外，想想 typedef static int int32 行不行？为什么？

下面再看一个与#define 宏有关的例子：

G)，#define PCHAR char*

PCHAR p3,p4;

H），typedef char* pchar;

pchar p1,p2;

两组代码编译都没有问题，但是，这里的 p4 却不是指针，仅仅是一个 char 类型的字符。这种错误很容易被忽略，所以用#define 的时候要慎之又慎。关于#define 当然还有很多话题需要讨论，请看预处理那一章。当然关于 typedef 的讨论也还没有结束，在指针与数组那一章，我们还要继续讨论。

1.17.4，#define a int[10]与 typedef int a[10];

留两个问题：

1），#define a int[10]

A),a[10] a[10];

B),a[10] a;

C),int a[10];

D),int a;

E),a b[10];

F),a b;

G),a* b[10];

H),a* b;

2), typedef int a[10];

A),a[10] a[10];

B),a[10] a;

C),int a[10];

D),int a;

E),a b[10];

F),a b;

G),a* b[10];

H),a* b;

3），#define a int*[10]

A),a[10] a[10];

B),a[10]   a;

C),int   a[10];

D),int   a;

E),a   b[10];

F),a   b;

G),a*   b[10];

H),a*   b;

4), typedef int * a[10];

A),a[10]   a[10];

B),a[10]   a;

C),int   a[10];

D),int   a;

E),a   b[10];

F),a   b;

G),a*   b[10];

H),a*   b;

5），#define   *a   int[10]

A),a[10]   a[10];

B),a[10]   a;

C),int   a[10];

D),int   a;

E),a   b[10];

F),a   b;

G),a*   b[10];

H),a*   b;

6), typedef int   (* a)[10];

A),a[10]   a[10];

B),a[10]   a;

C),int   a[10];

D),int   a;

E),a   b[10];

F),a   b;

G),a*　　b[10];

H),a*　　b;

7），#define　*a　* int[10]

A),a[10]　a[10];

B),a[10]　a;

C),int　　a[10];

D),int　　a;

E),a　　b[10];

F),a　　b;

G),a*　　b[10];

H),a*　　b;

8), typedef int *　(* a)[10];

A),a[10]　a[10];

B),a[10]　a;

C),int　　a[10];

D),int　　a;

E),a　　b[10];

F),a　　b;

G),a*　　b[10];

H),a*　　b;

请判断这里面哪些定义正确，哪些定义不正确。另外，int[10]和 a[10]到底该怎么用？

第二章 符号

符号有什么好说的呢？确实，符号可说的内容要少些，但总还是有些可以唠叨地方。有一次上课，我问学生：‘/’这个符号在 C 语言里都用在哪些地方？没有一个人能答完整。这说明 C 语言的基础掌握不牢靠，如果真正掌握了 C 语言，你就能很轻易的回答上来。这个问题就请读者试着回答一下吧。本章不会像关键字一样一个一个深入讨论，只是将容易出错的地方讨论一下。

表（2.1）标准 C 语言的基本符号

<table>
<tr><td>符号</td><td>名称</td><td>符号</td><td>名称</td></tr>
<tr><td>，</td><td>逗号</td><td>></td><td>右尖括号</td></tr>
<tr><td>.</td><td>圆点</td><td>!</td><td>感叹号</td></tr>
<tr><td>；</td><td>分号</td><td>|</td><td>竖线</td></tr>
<tr><td>：</td><td>冒号</td><td>/</td><td>斜杠</td></tr>
<tr><td>？</td><td>问号</td><td>\</td><td>反斜杠</td></tr>
<tr><td>’</td><td>单引号</td><td>~</td><td>波折号</td></tr>
<tr><td>“</td><td>双引号</td><td>#</td><td>井号</td></tr>
<tr><td>(</td><td>左圆括号</td><td>)</td><td>右圆括号</td></tr>
<tr><td>[</td><td>左方括号</td><td>]</td><td>右方括号</td></tr>
<tr><td>{</td><td>左大括号</td><td>}</td><td>右大括号</td></tr>
<tr><td>%</td><td>百分号</td><td>&</td><td>and（与）</td></tr>
<tr><td>^</td><td>xor（异或）</td><td>*</td><td>乘号</td></tr>
<tr><td>-</td><td>减号</td><td>=</td><td>等于号</td></tr>
<tr><td><</td><td>左尖括号</td><td>+</td><td>加号</td></tr>
</table>

C 语言的基本符号就有 20 多个，每个符号可能同时具有多重含义，而且这些符号之间相互组合又使得 C 语言中的符号变得更加复杂起来。

你也许听说过“国际 C 语言乱码大赛（IOCCC）”，能获奖的人毫无疑问是世界顶级 C 程序员。这是他们利用 C 语言的特点极限挖掘的结果。下面这个例子就是网上广为流传的一个经典作品：

```
#i nclude <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
```

```
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
  :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m.vpbks,fxntdCeghiry"),a+1);}
```

还没发狂？看来你抵抗力够强的。这是 IOCCC 1988 年获奖作品，作者是 Ian Phillipps。毫无疑问，Ian Phillipps 是世界上最顶级的 C 语言程序员之一。你可以数数这里面用了多少个符号。当然这里我并不会讨论这段代码，也并不是鼓励你也去写这样的代码（关于这段代码的分析，你可以上网查询）。恰恰相反，我要告诉你的是：

大师把代码写成这样是经典，你把代码写成这样是垃圾！

所以在垃圾和经典之间，你需要做一个抉择。

2.1，注释符号

2.1.1，几个似非而是的注释问题

C 语言的注释可以出现在 C 语言代码的任何地方。这句话对不对？这是我当学生时我老师问的一个问题。我当时回答是不对。好，那我们就看看下面的例子：

A），int/*...*/i;

B），char* s="abcdefgh   //hijklmn";

C), //Is it a \
  valid comment?

D), in/*…*/t i;

我们知道 C 语言里可以有两种注释方式：/* */ 和//。那上面 3 条注释对不对呢？建议你亲自在编译器中测试一下。上述前 3 条注释都是正确的，最后一条不正确。

A),有人认为编译器剔除掉注释后代码会被解析成 inti，所以不正确。编译器的确会将注释剔除，但不是简单的剔除，而是用空格代替原来的注释。再看一个例子：

/*这是*/#/*一条*/define/*合法的*/ID/*预处理*/replacement/*指*/list/*令*/

你可以用编译器试试。

B),我们知道双引号引起来的都是字符串常量，那双斜杠也不例外。

C),这是一条合法的注释，因为\是一个接续符。关于接续符，下面还有更多讨论。

D), 前面说过注释会被空格替换，那这条注释不正确就很好理解了。

现在你可以回答前面的问题了吧？

但注意：/*…*/这种形式的注释不能嵌套，如：

/*这是/*非法的*/*/

因为/*总是与离它最近的*/匹配。

2.1.2，y = x/*p

y = x/*p，这是表示 x 除以 p 指向的内存里的值，把结果赋值为 y？我们可以在编译器上测试一下，编译器提示出错。

实际上，编译器把/*当作是一段注释的开始，把/*后面的内容都当作注释内容，直到出现*/为止。这个表达式其实只是表示把 x 的值赋给 y，/*后面的内容都当作注释。但是，由于没有找到*/，所以提示出错。

我们可以把上面的表达式修改一下：

y = x/ *p

或者

y = x/(*p)

这样的话，表达式的意思就是 x 除以 p 指向的内存里的值，把结果赋值为 y 了。

也就是说只要斜杠（/）和星号（*）之间没有空格，都会被当作注释的开始。这一点一定要注意。

2.1.3，怎样才能写出出色的注释

注释写得出色非常不容易，但是写得糟糕却是人人可为之。糟糕的注释只会帮倒忙。

2.1.3.1，安息吧，路德维希.凡.贝多芬

在《Code Complete》这本书中，作者记录了这样一个故事：

有位负责维护的程序员半夜被叫起来，去修复一个出了问题的程序。但是程序的原作者已经离职，没有办法联系上他。这个程序员从未接触过这个程序。在仔细检查所有的说明后，他只发现了一条注释，如下：

MOV AX 723h ;R . I . P . L . V . B .

这个维护程序员通宵研究这个程序，还是对注释百思不得其解。虽然最后他还是把程序的问题成功排除了，但这个神秘的注释让他耿耿于怀。说明一点：汇编程序的注释是以分号开头。

几个月后，这名程序员在一个会议上遇到了注释的原作者。经过请教后，才明白这条注释的意思：安息吧，路德维希.凡.贝多芬（Rest in peace, Ludwig Van Neethoven）。贝多芬于 1827 年逝世，而 1827 的十六进制正是 723。这真是让人哭笑不得！

2.1.3.2，windows 大师们用注释讨论天气问题

还有个例子：前些日子 windows 的源代码曾经泄漏过一部分。人们在看这部分大师的

经典作品时，却发现很多与代码毫无关系的注释！有的注释在讨论天气，有的在讨论明天吃什么，还有的在骂公司和老板。这些注释虽然与代码无关，但总比上面那个让贝多芬安息的注释要强些的。至少不会让你抓狂。不过这种事情只有大师们才可以做，你可千万别用注释讨论天气。

2.1.3.3，出色注释的基本要求

【规则 2-1】注释应当准确、易懂，防止有二义性。错误的注释不但无益反而有害。

【规则 2-2】边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要及时删除。

【规则 2-3】注释是对代码的“提示”，而不是文档。程序中的注释应当简单明了，注释太多了会让人眼花缭乱。

【规则 2-4】一目了然的语句不加注释。

例如：i++; /* i 加 1 */

多余的注释

【规则 2-5】对于全局数据（全局变量、常量定义等）必须要加注释。

【规则 2-6】注释采用英文，尽量避免在注释中使用缩写，特别是不常用缩写。

因为不一定所有的编译器都能显示中文，别人打开你的代码，你的注释也许是一团乱码。还有，你的代码不一定是懂中文的人阅读。

【规则 2-7】注释的位置应与被描述的代码相邻，可以与语句在同一行，也可以在上行，但不可放在下方。同一结构中不同域的注释要对齐。

【规则 2-8】当代码比较长，特别是有多重嵌套时，应当在一些段落的结束处加注释，便于阅读。

【规则 2-9】注释的缩进要与代码的缩进一致。

【规则 2-10】注释代码段时应注重“为何做（why）”，而不是“怎么做（how）”。

说明怎么做的注释一般停留在编程语言的层次，而不是为了说明问题。尽力阐述“怎么做”的注释一般没有告诉我们操作的意图，而指明“怎么做”的注释通常是冗余的。

【规则 2-11】数值的单位一定要注释。

注释应该说明某数值的单位到底是什么意思。比如：关于长度的必须说明单位是毫米，米，还是千米等；关于时间的必须说明单位是时，分，秒，还是毫秒等。

【规则 2-12】对变量的范围给出注释。

【规则 2-13】对一系列的数字编号给出注释，尤其在编写底层驱动程序的时候（比如管脚编号）。

【规则 2-13】对于函数的入口出口数据给出注释。

关于函数的注释在函数那章有更详细的讨论。

2.2，接续符和转义符

C 语言里以反斜杠（\）表示断行。编译器会将反斜杠剔除掉，跟在反斜杠后面的字符自动接续到前一行。但是注意：反斜杠之后不能有空格，反斜杠的下一行之前也不能有空格。当然你可以测试一下加了空格之后的效果。我们看看下面的例子：

```
//这是一条合法的\
单行注释

/\
/这是一条合法的单行注释

#def\
ine MAC\
RO  这是一条合法的\
宏定义

cha\
r* s="这是一个合法的\\
n 字符串";
```

反斜杠除了可以被用作接续符，还能被用作转义字符的开始标识。

常用的转义字符及其含义：

转义字符	转义字符的意义
\n	回车换行
\t	横向跳到下一制表位置
\v	竖向跳格
\b	退格
\r	回车
\f	走纸换页
\\	反斜扛符"\"
\'	单引号符
\a	鸣铃
\ddd	1～3 位八进制数所代表的字符
\xhh	1～2 位十六进制数所代表的字符

广义地讲，C 语言字符集中的任何一个字符均可用转义字符来表示。表中的\ddd 和\xhh 正是为此而提出的。ddd 和 hh 分别为八进制和十六进制的 ASCII 代码。如\102 表示字母"B"，\134 表示反斜线，\X0A 表示换行等。

2.3，单引号、双引号

我们知道双引号引起来的都是字符串常量，单引号引起来的都是字符常量。但初学者还是容易弄错这两点。比如：‘a’和“a”完全不一样，在内存里前者占 1 个 byte，后者占2 个 byte。关于字符串常量在指针与数组那章将有更多的讨论。

这两个列子还好理解，再看看这三个：

1，‘1‘，“1”。

第一个是整形常数，32 位系统下占 4 个 byte；

第二个是字符常量，占 1 个 byte；

第三个是字符串常量，占 2 个 byte。

三者表示的意义完全不一样，所占的内存大小也不一样，初学者往往弄错。

字符在内存里是以 ASCAII 码存储的，所以字符常量可以与整形常量或变量进行运算。如：‘A‘ + 1。

2.4，逻辑运算符

||和&&是我们经常用到的逻辑运算符，与按位运算符|和&是两码事。下一节会介绍按位运算符。虽然简单，但毕竟容易犯错。看例子：

```
int i=0;
int j=0;
if((++i>0)||(++j>0))
{
    //打印出 i 和 j 的值。
}
```

结果:i=1;j=0。

不要惊讶。逻辑运算符||两边的条件只要有一个为真，其结果就为真；只要有一个结果为假，其结果就为假。if((++i>0)||(++j>0))语句中，先计算(++i>0)，发现其结果为真，后面的(++j>0)便不再计算。同样&&运算符也要注意这种情况。这是很容易出错的地方，希望读者注意。

2.5，位运算符

C 语言中位运算包括下面几种：

& 按位与
| 按位或
^ 按位异或
~ 取反
<< 左移
>> 右移

前 4 种操作很简单，一般不会出错。但要注意按位运算符|和&与逻辑运算符||和&&完全是两码事，别混淆了。其中按位异或操作可以实现不用第三个临时变量交换两个变量的值：a ^= b; b ^= a;a ^= b;但并不推荐这么做，因为这样的代码读起来很费劲。

2.5.1，左移和右移

下面讨论一下左移和右移：

左移运算符“<<”是双目运算符。其功能把“<< ”左边的运算数的各二进位全部左移若干位，由“<<”右边的数指定移动的位数，高位丢弃，低位补 0。

右移运算符“>>”是双目运算符。其功能是把“>> ”左边的运算数的各二进位全部右移若干位，“>>”右边的数指定移动的位数。但注意：对于有符号数，在右移时，符号位将随同移动。当为正数时， 最高位补 0；而为负数时，符号位为 1，最高位是补 0 或是补 1 取决于编译系统的规定。Turbo C 和很多系统规定为补 1。

2.5.2，0x01<<2+3 的值为多少？

再看看下面的例子：

0x01<<2+3；

结果为 7 吗？测试一下。结果为 32？别惊讶，32 才是正确答案。因为“+”号的优先级比移位运算符的优先级高（关于运算符的优先级，我并不想在这里做过多的讨论，你几乎可以在任何一本 C 语言书上找到）。好，在 32 位系统下，再把这个例子改写一下：

0x01<<2+30；或 0x01<<2-3；

这样行吗？不行。一个整型数长度为 32 位，左移 32 位发生了什么事情？溢出！左移-1 位呢？反过来移？所以，左移和右移的位数是有讲究的。**左移和右移的位数不能大于数据的长度，不能小于 0**。

2.6，花括号

花括号每个人都见过，很简单吧。但曾经有一个学生问过我如下问题：

char　a[10] = {“abcde”};

他不理解为什么这个表达式正确。我让他继续改一下这个例子：

char　a[10] { = “abcde”};

问他这样行不行。那读者以为呢？为什么？

花括号的作用是什么呢？我们平时写函数，if、while、for、switch 语句等都用到了它，但有时又省略掉了它。简单来说花括号的作用就是打包。你想想以前用花括号是不是为了把一些语句或代码打个包包起来，使之形成一个整体，并与外界绝缘。这样理解的话，上面的问题就不是问题了。

2.7，++、--操作符

这绝对是一对让人头疼的兄弟。先来点简单的：

int i = 3；

（++i）+（++i）+（++i）;

表达式的值为多少？15 吗？16 吗？18 吗？其实对于这种情况，C 语言标准并没有作出规定。有点编译器计算出来为 18，因为 i 经过 3 次自加后变为 6，然后 3 个 6 相加得 18；而有的编译器计算出来为 16（比如 Visual C++6.0），先计算前两个 i 的和，这时候 i 自加两次，2 个 i 的和为 10，然后再加上第三次自加的 i 得 16。其实这些没有必要辩论，用到哪个编译器写句代码测试就行了。但不会计算出 15 的结果来的。

++、--作为前缀，我们知道是先自加或自减，然后再做别的运算；但是作为后缀时，到底什么时候自加、自减？这是很多初学者迷糊的地方。假设 i=0，看例子：

```
A),j =(i++,i++,i++);
B),for（i=0;i<10;i++）
  {
      //code
  }
C)，k = （i++）+ （i++）+ （i++）;
```

你可以试着计算他们的结果。

A) 例子为逗号表达式，i 在遇到每个逗号后，认为本计算单位已经结束，i 这时候自加。

关于逗号表达式与“++”或“--”的连用，还有一个比较好的例子：

```
int x;
int i = 3;
```

x = (++i, i++, i+10);

问 x 的值为多少？i 的值为多少？

按照上面的讲解，可以很清楚的知道，逗号表达式中，i 在遇到每个逗号后，认为本计算单位已经结束，i 这时候自加。所以，本例子计算完后， i 的值为 5， x 的值为 15。

B) 例子 i 与 10 进行比较之后，认为本计算单位已经结束，i 这时候自加。

C) 例子 i 遇到分号才认为本计算单位已经结束，i 这时候自加。

也就是说后缀运算是在本计算单位计算结束之后再自加或自减。C 语言里的计算单位大体分为以上 3 类。

留一个问题：

```
for（i=0，printf（“First=%d”，i）;
    i<10，printf（“Second=%d”，i）;
    i++，printf（“Third=%d”，i））
{
    printf（“Fourth=%d”，i);
}
```

打印出什么结果？

2.7.1，++i+++i+++i

上面的例子很简单，那我们把括号去掉看看：

```
int i = 3;
++i+++i+++i;
```

天啦！这到底是什么东西？好，我们先看看这个：a+++b 和下面哪个表达式想当：

A),a++ +b;

B),a+ ++b;

2.7.2，贪心法

C 语言有这样一个规则：每一个符号应该包含尽可能多的字符。也就是说，编译器将程序分解成符号的方法是，从左到右一个一个字符地读入，如果该字符可能组成一个符号，那么再读入下一个字符，判断已经读入的两个字符组成的字符串是否可能是一个符号的组成部分；如果可能，继续读入下一个字符，重复上述判断，直到读入的字符组成的字符串已不再可能组成一个有意义的符号。这个处理的策略被称为“贪心法”。需要注意到是，除了字符串与字符常量，符号的中间不能嵌有空白（空格、制表符、换行符等）。比如：==是

单个符号，而= =是两个等号。

按照这个规则可能很轻松的判断 a+++b 表达式与 a++ +b 一致。那++i+++i+++i；会被解析成什么样子呢？希望读者好好研究研究。另外还可以考虑一下这个表达式的意思：a+++++b；

2.8，2/(-2)的值是多少？

除法运算在小学就掌握了的，这里还要讨论什么呢？别急，先计算下面这个例子：

2/(-2)的值为多少？2%(-2)的值呢？

如果与你想象的结果不一致，不要惊讶。我们先看看下面这些规则：

假定我们让 a 除以 b，商为 q，余数为 r：

q = a/b;

r = a%b;

这里不妨先假定 b 大于 0。

我们希望 a、b、q、r 之间维持什么样的关系呢？

1，最重要的一点，我们希望 q*b + r == a，因为这是定义余数的关系。

2，如果我们改变 a 的正负号，我们希望 q 的符号也随之改变，但 q 的绝对值不会变。

3，当 b>0 时，我们希望保证 r>=0 且 r<b。

这三条性质是我们认为整数除法和余数操作所应该具备的。但是，很不幸，它们不可能同时成立。

先考虑一个简单的例子：3/2，商为 1，余数也为 1。此时，第一条性质得到了满足。

好，把例子稍微改写一下：(-3)/2 的值应该是多少呢？如果要满足第二条性质，答案应该是-1。但是，如果是这样，余数就必定是-1，这样第三条性质就无法满足了。如果我们首先满足第三条性质，即余数是 1，这种情况下根据第一条性质，商应该为-2，那么第二条性质又无法满足了。

上面的矛盾似乎无法解决。因此，C 语言或者其他语言在实现整数除法截断运算时，必须放弃上述三条性质中的至少一条。大多数编程语言选择了放弃第三条，而改为要求余数与被除数的正负号相同。这样性质 1 和性质 2 就可以得到满足。大多数 C 语言编译器也都是如此。

但是，C 语言的定义只保证了性质 1，以及当 a>=0 且 b>0 时，保证|r|<|b|以及 r>=0。后面部分的保证与性质 2 或性质 3 比较起来，限制性要弱得多。

通过上面的解释，你是否能准确算出 2/(-2)和 2%(-2)的值呢？

2.9，运算符的优先级

2.9.1，运算符的优先级表

C 语言的符号众多，由这些符号又组合成了各种各样的运算符。既然是运算符就一定有其特定的优先级，下表就是 C 语言运算符的优先级表：

优先级	运算符	名称或含义	使用形式	结合方向	说明

1	[]	数组下标	数组名[常量表达式]	左到右	
	()	圆括号	(表达式)/函数名(形参表)		
	.	成员选择（对象）	对象.成员名		
	->	成员选择（指针）	对象指针->成员名		
2	-	负号运算符	-表达式	右到左	单目运算符
	(类型)	强制类型转换	(数据类型)表达式		
	++	自增运算符	++变量名/变量名++		单目运算符
	--	自减运算符	--变量名/变量名--		单目运算符
	*	取值运算符	*指针变量		单目运算符
	&	取地址运算符	&变量名		单目运算符
	!	逻辑非运算符	!表达式		单目运算符
	~	按位取反运算符	~表达式		单目运算符
	sizeof	长度运算符	sizeof(表达式)		
3	/	除	表达式/表达式	左到右	双目运算符
	*	乘	表达式*表达式		双目运算符
	%	余数（取模）	整型表达式/整型表达式		双目运算符
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式		双目运算符
5	<<	左移	变量<<表达式	左到右	双目运算符
	>>	右移	变量>>表达式		双目运算符
6	>	大于	表达式>表达式	左到右	双目运算符
	>=	大于等于	表达式>=表达式		双目运算符
	<	小于	表达式<表达式		双目运算符
	<=	小于等于	表达式<=表达式		双目运算符
7	==	等于	表达式==表达式	左到右	双目运算符
	!=	不等于	表达式!= 表达式		双目运算符
8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10	|	按位或	表达式|表达式	左到右	双目运算符
11	&&	逻辑与	表达式&&表达式	左到右	双目运算符
12	||	逻辑或	表达式||表达式	左到右	双目运算符
13	?:	条件运算符	表达式 1? 表达式 2: 表达式 3	右到左	三目运算符
14	=	赋值运算符	变量=表达式	右到左	
	/=	除后赋值	变量/=表达式		
	*=	乘后赋值	变量*=表达式		
	%=	取模后赋值	变量%=表达式		
	+=	加后赋值	变量+=表达式		
	-=	减后赋值	变量-=表达式		
	<<=	左移后赋值	变量<<=表达式		
	>>=	右移后赋值	变量>>=表达式		

	&=	按位与后赋值	变量&=表达式		
	^=	按位异或后赋值	变量^=表达式		
	|=	按位或后赋值	变量|=表达式		
15	,	逗号运算符	表达式,表达式,…	左到右	从左向右顺序运算

注：同一优先级的运算符，运算次序由结合方向所决定。

上表不容易记住。其实也用不着死记，用得多，看得多自然就记得了。也有人说不用记这些东西，只要记住乘除法的优先级比加减法高就行了，别的地方一律加上括号。这在你自己写代码的时候，确实可以，但如果是你去阅读和理解别人的代码呢？别人不一定都加上括号了吧？所以，记住这个表，我个人认为还是很有必要的。

2.9.2，一些容易出错的优先级问题

上表中，优先级同为 1 的几种运算符如果同时出现，那怎么确定表达式的优先级呢？这是很多初学者迷糊的地方。下表就整理了这些容易出错的情况：

优先级问题	表达式	经常误认为的结果	实际结果
. 的优先级高于* ->操作符用于消除这个问题	*p.f	p 所指对象的字段 f (*p).f	对 p 取 f 偏移，作为指针，然后进行解除引用操作。*(p.f)
[]高于*	int *ap[]	ap 是个指向 int 数组的指针 int (*ap)[]	ap 是个元素为 int 指针的数组 int *(ap[])
函数()高于*	int *fp()	fp 是个函数指针，所指函数返回 int。 int (*fp)()	fp 是个函数，返回 int * int *(fp())
== 和!=高于位操作	(val & mask != 0)	(val & mask)!= 0	val & (mask != 0)
== 和!=高于赋值符	c = getchar() != EOF	(c = getchar()) != EOF	c = (getchar() != EOF)
算术运算符高于位移运算符	msb << 4 + lsb	(msb << 4) + lsb	msb << (4 + lsb)
逗号运算符在所有运算符中优先级最低	i = 1,2	i = (1,2)	(i = 1),2

这些容易出错的情况，希望读者好好在编译器上调试调试，这样印象会深一些。一定要多调试，光靠看代码，水平是很难提上来的。调试代码才是最长水平的。

第三章 预处理

往往我说今天上课的内容是预处理时，便有学生质疑：预处理不就是 include 和 define 么？这也用得着讲啊？。是的，非常值得讨论，即使是 include 和 define。但是预处理仅限于此吗？远远不止。先看几个个常识性问题：

A),预处理是 C 语言的一部分吗？

B),包含“#”号的都是预处理吗？

C),预处理指令后面都不需要加“；”号吗？

不要急着回答，先看看 ANSI 标准定义的 C 语言预处理指令：

表（3.1） 预处理指令

预处理名称	意 义
#define	宏定义
#undef	撤销已定义过的宏名
#include	使编译程序将另一源文件嵌入到带有#include 的源文件中
#if #else #elif #endif	#if 的一般含义是如果#if 后面的常量表达式为 true，则编译它与#endif 之间的代码，否则跳过这些代码。命令#endif 标识一个#if 块的结束。#else 命令的功能有点象 C 语言中的 else ，#else 建立另一选择（在# if 失败的情况下）。#elif 命令意义与 else if 相同，它形成一个 if else-if 阶梯状语句，可进行多种编译选择。
#ifdef #ifndef	用#ifdef 与#ifndef 命令分别表示“如果有定义”及“如果无定义”，是条件编译的另一种方法。
#line	改变当前行数和文件名称，它们是在编译程序中预先定义的标识符 命令的基本形式如下： #line number["filename"]
#error	编译程序时，只要遇到 #error 就会生成一个编译错误提示消息，并停止编译
#pragma	为实现时定义的命令，它允许向编译程序传送各种指令例如，编译程序可能有一种选择，它支持对程序执行的跟踪。可用#pragma 语句指定一个跟踪选择。

另外 ANSI 标准 C 还定义了如下几个宏：

LINE 表示正在编译的文件的行号

FILE 表示正在编译的文件的名字

DATE 表示编译时刻的日期字符串，例如： "25 Dec 2007"

TIME 表示编译时刻的时间字符串，例如： "12:30:55"

STDC 判断该文件是不是定义成标准 C 程序

如果编译器不是标准的，则可能仅支持以上宏的一部分，或根本不支持。当然编译器也有可能还提供其它预定义的宏名。注意：宏名的书写由标识符与两边各二条下划线构成。

相信很多初学者，甚至一些有经验的程序员都没有完全掌握这些内容，下面就一一详细讨论这些预处理指令。

3.1，宏定义

3.1.1，数值宏常量

#define 宏定义是个演技非常高超的替身演员，但也会经常耍大牌的，所以我们用它要慎之又慎。它可以出现在代码的任何地方，**从本行宏定义开始**，以后的代码就就都认识这个宏了；也可以把任何东西定义成宏。因为编译器会在预编译的时候用真身替换替身，而在我们的代码里面却又用常常用替身来帮忙。看例子：

```
#define    PI    3.141592654
```

在此后的代码中你尽可以使用 PI 来代替 3.141592654，而且你最好就这么做。不然的话，如果我要把 PI 的精度再提高一些，你是否愿意一个一个的去修改这串数呢？你能保证不漏不出错？而使用 PI 的话，我们却只需要修改一次。这种情况还不是最要命的，我们再看一个例子：

```
#define      ERROR_POWEROFF    -1
```

如果你在代码里不用 ERROR_POWEROFF 这个宏而用-1，尤其在函数返回错误代码的时候（往往一个开发一个系统需要定义很多错误代码）。肯怕上帝都无法知道-1 表示的是什么意思吧。这个-1，我们一般称为“魔鬼数”，上帝遇到它也会发狂的。所以，我奉劝你代码里一定不要出现“魔鬼数”。

第一章我们详细讨论了 const 这个关键字，我们知道 const 修饰的数据是有类型的，而 define 宏定义的数据没有类型。为了安全，我建议你以后在定义一些宏常数的时候用 const 代替，编译器会给 const 修饰的只读变量做类型校验，减少错误的可能。但一定要注意 const 修饰的不是常量而是 readonly 的变量，**const 修饰的只读变量不能用来作为定义数组的维数，也不能放在 case 关键字后面。**

3.1.2，字符串宏常量

除了定义宏常数之外，经常还用来定义字符串，尤其是路径：

A),#define   ENG_PATH_1   E:\English\listen_to_this\listen_to_this_3

B),#define   ENG_PATH_2   “E:\English\listen_to_this\listen_to_this_3”

噢，到底哪一个正确呢？如果路径太长，一行写下来比较别扭怎么办？用反斜杠接续符啊：

C), #define ENG_PATH_3 E:\English\listen_to_this\listen\

_to_this_3

还没发现问题？这里用了 4 个反斜杠，到底哪个是接续符？回去看看接续符反斜杠。反斜杠作为接续符时，在本行其后面不能再有任何字符，空格都不行。所以，只有最后一个反斜杠才是接续符。至于 A)和 B)，那要看你怎么用了，既然 define 宏只是简单的替换，那给 ENG_PATH_1 加上双引号不就成了："ENG_PATH_1"。

但是请注意：有的系统里规定路径的要用双反斜杠"\\",比如：

#define ENG_PATH_4   E:\\English\\listen_to_this\\listen_to_this_3

3.1.3，用 define 宏定义注释符号？

上面对 define 的使用都很简单，再看看下面的例子：

#define BSC //

#define BMC /*

#define EMC */

D),BSC my single-line comment

E),BMC my multi-line comment EMC

D)和 E)都错误，为什么呢？因为注释先于预处理指令被处理,当这两行被展开成//…或/*…*/时,注释已处理完毕,此时再出现//…或/*…*/自然错误.因此,试图用宏开始或结束一段注释是不行的。

3.1.4，用 define 宏定义表达式

这些都好理解，下面来点有"技术含量"的：

定义一年有多少秒：

#define SEC_A_YEAR 60*60*24*365

这个定义没错吧？很遗憾，很有可能错了，至少不可靠。你有没有考虑在 16 位系统下把这样一个数赋给整型变量的时候可能会发生溢出？一年有多少秒也不可能是负数吧。修改一下：

#define SEC_A_YEAR （60*60*24*365）UL

又出现一个问题，这里的括号到底需不需要呢？继续看一个例子：

定义一个宏函数，求 x 的平方：

#define　SQR (x)　x * x

对不对？试试：假设 x 的值为 10，SQR (x)被替换后变成 10*10。没有问题。

再试试：假设 x 的值是个表达式 10+1，SQR (x)被替换后变成 10+1*10+1。问题来了，这并不是我想要得到的。怎么办？括号括起来不就完了？

#define　SQR (x)　（(x）*（x））

最外层的括号最好也别省了，看例子：

求两个数的和：

#define　SUM (x)　（x）+（x）

如果 x 的值是个表达式 5*3,而代码又写成这样：SUM (x)* SUM (x)。替换后变成：（5*3）+（5*3）*（5*3）+（5*3）。又错了！所以最外层的括号最好也别省了。我说过 define 是个演技高超的替身演员，但也经常耍大牌。**要搞定它其实很简单，别吝啬括号就行了。**

注意这一点：宏函数被调用时是以实参代换形参。而不是“值传送”。

留四个问题：

A)，上述宏定义中“SUM”、“SQR”是宏吗？

B)，#define EMPTY

　　这样定义行吗？

C)，打印上述宏定义的值：printf（“SUM (x)”）；结果是什么？

D), “#define M 100”是宏定义吗？

3.1.5，宏定义中的空格

另外还有一个问题需要引起注意，看下面例子：

#define　SUM　（x）　（x）+（x）

这还是定义的宏函数 SUM（x）吗？显然不是。编译器认为这是定义了一个宏：SUM，其代表的是（x）　（x）+（x）。为什么会这样呢？其关键问题还是在于 SUM 后面的这个空格。所以在定义宏的时候一定要注意什么时候该用空格，什么时候不该用空格。这个空格仅仅在定义的时候有效，在使用这个宏函数的时候，空格会被编译器忽略掉。也就是说，上一节定义好的宏函数 SUM（x）在使用的时候在 SUM 和（x）之间留有空格是没问题的。比如：SUM（3）和 SUM　（3）的意思是一样的。

3.1.6，#undef

#undef 是用来撤销宏定义的，用法如下：

#define　PI　3.141592654

…

// code

#undef　PI

//下面的代码就不能用 PI 了，它已经被撤销了宏定义。

也就是说宏的生命周期从#define 开始到#undef 结束。很简单，但是请思考一下这个问题：

#define　X　3

#define　Y　X*2

#undef　X

#define　X　2

int z=Y;

z 的值为多少？

3.2，条件编译

条件编译的功能使得我们可以按不同的条件去编译不同的程序部分，因而产生不同的目标代码文件。这对于程序的移植和调试是很有用的。条件编译有三种形式，下面分别介绍：

第一种形式：
#ifdef 标识符
程序段 1
#else
程序段 2
#endif

它的功能是，如果标识符已被 #define 命令定义过则对程序段 1 进行编译；否则对程序段 2 进行编译。如果没有程序段 2(它为空)，本格式中的#else 可以没有，即可以写为：
#ifdef 标识符
程序段

#endif

第二种形式：

#ifndef 标识符
程序段 1
#else
程序段 2
#endif

与第一种形式的区别是将“ifdef”改为“ifndef”。它的功能是，如果标识符未被#define 命令定义过则对程序段 1 进行编译，否则对程序段 2 进行编译。这与第一种形式的功能正相反。

第三种形式：

#if 常量表达式
程序段 1

#else

程序段 2

#endif

它的功能是，如常量表达式的值为真(非 0)，则对程序段 1 进行编译，否则对程序段 2 进行编译。因此可以使程序在不同条件下，完成不同的功能。

至于#elif 命令意义与 else if 相同，它形成一个 if else-if 阶梯状语句，可进行多种编译选择。

3.3，文件包含

文件包含是预处理的一个重要功能，它可用来把多个源文件连接成一个源文件进行编译，结果将生成一个目标文件。C 语言提供#include 命令来实现文件包含的操作，它实际是宏替换的延伸，有两种格式：

格式 1：

#include <filename>

其中，filename 为要包含的文件名称，用尖括号括起来，也称为头文件，表示预处理到系统规定的路径中去获得这个文件（即 C 编译系统所提供的并存放在指定的子目录下的头文件）。找到文件后，用文件内容替换该语句。

格式 2：

#include “filename”

其中，filename 为要包含的文件名称。双引号表示预处理应在当前目录中查找文件名为 filename 的文件，若没有找到，则按系统指定的路径信息，搜索其他目录。找到文件后，用文件内容替换该语句。

需要强调的一点是：#include 是将已存在文件的内容嵌入到当前文件中。

另外关于#include 的路径也有点要说明：include 支持相对路径，格式如 trackant(蚁迹寻踪)所写：

.代表当前目录，..代表上层目录。

3.4，#error 预处理

#error 预处理指令的作用是，编译程序时，只要遇到 #error 就会生成一个编译错误提示消息，并停止编译。其语法格式为：

#error error-message

注意，宏串 error-message 不用双引号包围。遇到#error 指令时，错误信息被显示，可能同时还显示编译程序作者预先定义的其他内容。关于系统所支持的 error-message 信息，请查找相关资料，这里不浪费篇幅来做讨论。

3.5，#line 预处理

#line 的作用是改变当前行数和文件名称，它们是在编译程序中预先定义的标识符

命令的基本形式如下：

#line number["filename"]

其中[]内的文件名可以省略。

例如：

#line 30 a.h

其中，文件名 a.h 可以省略不写。

这条指令可以改变当前的行号和文件名，例如上面的这条预处理指令就可以改变当前的行号为 30，文件名是 a.h。初看起来似乎没有什么用，不过，他还是有点用的，那就是用在编译器的编写中，我们知道编译器对 C 源码编译过程中会产生一些中间文件，通过这条指令，可以保证文件名是固定的，不会被这些中间文件代替，有利于进行分析。

3.6，#pragma 预处理

在所有的预处理指令中，#pragma 指令可能是最复杂的了，它的作用是设定编译器的状态或者是指示编译器完成一些特定的动作。#pragma 指令对每个编译器给出了一个方法,在保持与 C 和 C ++语言完全兼容的情况下,给出主机或操作系统专有的特征。依据定义,编译指示是机器或操作系统专有的,且对于每个编译器都是不同的。

其格式一般为:

#pragma para

其中 para 为参数，下面来看一些常用的参数。

3.6.1，#pragma message

message 参数：Message 参数是我最喜欢的一个参数，它能够在编译信息输出窗口中输出相应的信息，这对于源代码信息的控制是非常重要的。其使用方法为：

#pragma message(“消息文本”)

当编译器遇到这条指令时就在编译输出窗口中将消息文本打印出来。

当我们在程序中定义了许多宏来控制源代码版本的时候，我们自己有可能都会忘记有没有正确的设置这些宏，此时我们可以用这条指令在编译的时候就进行检查。假设我们希望判断自己有没有在源代码的什么地方定义了_X86 这个宏可以用下面的方法

```
#ifdef _X86
#Pragma message(“_X86 macro activated!”)
#endif
```

当我们定义了_X86 这个宏以后，应用程序在编译时就会在编译输出窗口里显示“_X86 macro activated!”。我们就不会因为不记得自己定义的一些特定的宏而抓耳挠腮了

。

3.6.2，#pragma code_seg

另一个使用得比较多的 pragma 参数是 code_seg。格式如：

#pragma code_seg(["section-name"[,"section-class"]])

它能够设置程序中函数代码存放的代码段，当我们开发驱动程序的时候就会使用到它。

3.6.3，#pragma once

#pragma once (比较常用）

只要在头文件的最开始加入这条指令就能够保证头文件被编译一次，这条指令实际上在 Visual C++6.0 中就已经有了，但是考虑到兼容性并没有太多的使用它。

3.6.4，#pragma hdrstop

#pragma hdrstop 表示预编译头文件到此为止，后面的头文件不进行预编译。BCB 可以预编译头文件以加快链接的速度，但如果所有头文件都进行预编译又可能占太多磁盘空间，所以使用这个选项排除一些头文件。

有时单元之间有依赖关系，比如单元 A 依赖单元 B，所以单元 B 要先于单元 A 编译。你可以用#pragma startup 指定编译优先级，如果使用了#pragma package(smart_init) ，BCB 就会根据优先级的大小先后编译。

3.6.5，#pragma resource

#pragma resource "*.dfm"表示把*.dfm 文件中的资源加入工程。*.dfm 中包括窗体外观的定义。

3.6.6，#pragma warning

#pragma warning(disable : 4507 34; once : 4385; error : 164)

等价于：

#pragma warning(disable:4507 34) // 不显示 4507 和 34 号警告信息

#pragma warning(once:4385) // 4385 号警告信息仅报告一次

#pragma warning(error:164) // 把 164 号警告信息作为一个错误。

同时这个 pragma warning 也支持如下格式：

#pragma warning(push [,n])

#pragma warning(pop)

这里 n 代表一个警告等级(1---4)。

#pragma warning(push)保存所有警告信息的现有的警告状态。

#pragma warning(push, n)保存所有警告信息的现有的警告状态，并且把全局警告等级设定为 n。

#pragma warning(pop)向栈中弹出最后一个警告信息，在入栈和出栈之间所作的一切改动取消。例如：

#pragma warning(push)

#pragma warning(disable : 4705)

```
#pragma warning( disable : 4706 )
#pragma warning( disable : 4707 )
//.......
#pragma warning( pop )
```

在这段代码的最后，重新保存所有的警告信息(包括 4705，4706 和 4707)。

3.6.7，#pragma comment

```
#pragma comment(...)
```

该指令将一个注释记录放入一个对象文件或可执行文件中。

常用的 lib 关键字，可以帮我们连入一个库文件。 比如：

```
#pragma comment(lib, "user32.lib")
```

该指令用来将 user32.lib 库文件加入到本工程中。

linker:将一个链接选项放入目标文件中,你可以使用这个指令来代替由命令行传入的或者在开发环境中设置的链接选项,你可以指定/include 选项来强制包含某个对象,例如:

```
#pragma comment(linker, "/include:__mySymbol")
```

3.6.8，#pragma pack

这里重点讨论内存对齐的问题和#pragma pack（）的使用方法。

什么是内存对齐？

先看下面的结构：

```
struct TestStruct1
{
    char c1;
    short s;
    char c2;
    int i;
};
```

假设这个结构的成员在内存中是紧凑排列的，假设 c1 的地址是 0，那么 s 的地址就应该是 1，c2 的地址就是 3，i 的地址就是 4。也就是 c1 地址为 00000000, s 地址为 00000001, c2 地址为 00000003, i 地址为 00000004。

可是，我们在 Visual C++6.0 中写一个简单的程序：

```
struct TestStruct1 a;
printf("c1 %p, s %p, c2 %p, i %p\n",
   (unsigned int)(void*)&a.c1 - (unsigned int)(void*)&a,
   (unsigned int)(void*)&a.s - (unsigned int)(void*)&a,
   (unsigned int)(void*)&a.c2 - (unsigned int)(void*)&a,
   (unsigned int)(void*)&a.i - (unsigned int)(void*)&a);
```

运行，输出：

c1 00000000, s 00000002, c2 00000004, i 00000008。

为什么会这样？这就是内存对齐而导致的问题。

3.6.8.1，为什么会有内存对齐？

字，双字，和四字在自然边界上不需要在内存中对齐。（对字，双字，和四字来说，自然边界分别是偶数地址，可以被 4 整除的地址，和可以被 8 整除的地址。）无论如何，为了提高程序的性能，数据结构（尤其是栈）应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；然而，对齐的内存访问仅需要一次访问。

一个字或双字操作数跨越了 4 字节边界，或者一个四字操作数跨越了 8 字节边界，被认为是未对齐的，从而需要两次总线周期来访问内存。一个字起始地址是奇数但却没有跨越字边界被认为是对齐的，能够在一个总线周期中被访问。某些操作双四字的指令需要内存操作数在自然边界上对齐。如果操作数没有对齐，这些指令将会产生一个通用保护异常。双四字的自然边界是能够被 16 整除的地址。其他的操作双四字的指令允许未对齐的访问（不会产生通用保护异常），然而，需要额外的内存总线周期来访问内存中未对齐的数据。

缺省情况下，编译器默认将结构、栈中的成员数据进行内存对齐。因此，上面的程序输出就变成了：c1 00000000, s 00000002, c2 00000004, i 00000008。编译器将未对齐的成员向后移，将每一个都成员对齐到自然边界上，从而也导致了整个结构的尺寸变大。尽管会牺牲一点空间（成员之间有部分内存空闲），但提高了性能。也正是这个原因，我们不可以断言 sizeof(TestStruct1)的结果为 8。在这个例子中，sizeof(TestStruct1)的结果为 12。

3.6.8.2，如何避免内存对齐的影响

那么，能不能既达到提高性能的目的，又能节约一点空间呢？有一点小技巧可以使用。比如我们可以将上面的结构改成：

```
struct TestStruct2
{
    char c1;
    char c2;
    short s;
    int i;
};
```

这样一来，每个成员都对齐在其自然边界上，从而避免了编译器自动对齐。在这个例子中，sizeof(TestStruct2)的值为 8。这个技巧有一个重要的作用，尤其是这个结构作为 API 的一部分提供给第三方开发使用的时候。第三方开发者可能将编译器的默认对齐选项改变，从而造成这个结构在你的发行的 DLL 中使用某种对齐方式，而在第三方开发者哪里却使用另外一种对齐方式。这将会导致重大问题。

比如，TestStruct1 结构，我们的 DLL 使用默认对齐选项，对齐为

c1 00000000, s 00000002, c2 00000004, i 00000008，同时 sizeof(TestStruct1)的值为 12。

而第三方将对齐选项关闭，导致

c1 00000000, s 00000001, c2 00000003, i 00000004，同时 sizeof(TestStruct1)的值为 8。

除此之外我们还可以利用#pragma pack（）来改变编译器的默认对齐方式（当然一般编译器

也提供了一些改变对齐方式的选项，这里不讨论）。

使用指令#pragma pack (n)，编译器将按照 n 个字节对齐。
使用指令#pragma pack ()，编译器将取消自定义字节对齐方式。

在#pragma pack (n)和#pragma pack ()之间的代码按 n 个字节对齐。

但是，成员对齐有一个重要的条件,即每个成员按自己的方式对齐.也就是说虽然指定了按 n 字节对齐,但并不是所有的成员都是以 n 字节对齐。其对齐的规则是,每个成员按其类型的对齐参数(通常是这个类型的大小)和指定对齐参数(这里是 n 字节)中较小的一个对齐，即：min(n, sizeof(item)) 。并且结构的长度必须为所用过的所有对齐参数的整数倍,不够就补空字节。看如下例子：

```
#pragma pack(8)

struct TestStruct4

{

    char    a;
    long    b;

};

struct TestStruct5

{

    char c;
    TestStruct4 d;
    long long e;

};

#pragma pack()
```

问题：

A),sizeof(TestStruct5) = ?
B), TestStruct5 的 c 后面空了几个字节接着是 d?

TestStruct4 中,成员 a 是 1 字节默认按 1 字节对齐,指定对齐参数为 8,这两个值中取 1,a 按 1 字节对齐;成员 b 是 4 个字节,默认是按 4 字节对齐,这时就按 4 字节对齐,所以 sizeof(TestStruct4)应该为 8;

TestStruct5 中,c 和 TestStruct4 中的 a 一样,按 1 字节对齐,而 d 是个结构,它是 8 个字节,它按什么对齐呢?对于结构来说,它的默认对齐方式就是它的所有成员使用的对齐参数中最大的一个, TestStruct4 的就是 4.所以,成员 d 就是按 4 字节对齐.成员 e 是 8 个字节,它是默认按 8 字节对齐,和指定的一样,所以它对到 8 字节的边界上,这时,已经使用了 12 个字节了,所以又添加了 4 个字节的空,从第 16 个字节开始放置成员 e.这时,长度为 24,已经可以被 8(成员 e 按 8 字节对齐)整除.这样,一共使用了 24 个字节.内存布局如下（*表示空闲内存，1 表示使用内存。单位为 1byete）：

```
                          a    b
TestStruct4 的内存布局：1***,1111,
                           c       TestStruct4.a    TestStruct4.b       d
TestStruct5 的内存布局：  1***,   1***,             1111,          ****，11111111
```

这里有三点很重要:

首先，每个成员分别按自己的方式对齐,并能最小化长度。

其次,复杂类型(如结构)的默认对齐方式是它最长的成员的对齐方式,这样在成员是复杂类型时,可以最小化长度。

然后，对齐后的长度必须是成员中最大的对齐参数的整数倍,这样在处理数组时可以保证每一项都边界对齐。

补充一下,对于数组,比如:char a[3];它的对齐方式和分别写 3 个 char 是一样的.也就是说它还是按 1 个字节对齐.如果写: typedef char Array3[3];Array3 这种类型的对齐方式还是按 1 个字节对齐,而不是按它的长度。

但是不论类型是什么,对齐的边界一定是 1,2,4,8,16,32,64....中的一个。

另外，注意别的#pragma pack 的其他用法：

#pragma pack(push)　　//保存当前对其方式到 packing stack

#pragma pack(push,n)　等效于

#pragma pack(push)

#pragma pack(n)　//n=1,2,4,8,16 保存当前对齐方式，设置按 n 字节对齐

#pragma pack(pop)　　//packing stack 出栈，并将对其方式设置为出栈的对齐方

3.7，#运算符

#也是预处理？是的，你可以这么认为。那怎么用它呢？别急，先看下面例子：

#define SQR(x)　printf("The square of　x　is %d.\n", ((x)*(x)));

如果这样使用宏：

SQR(8);

则输出为：

The square of　x　is 64.

注意到没有，引号中的字符 x 被当作普通文本来处理，而不是被当作一个可以被替换的语言符号。

假如你确实希望在字符串中包含宏参数，那我们就可以使用“#”，它可以把语言符号转化为字符串。上面的例子改一改：

#define SQR(x)　printf("The square of　"#x"　is %d.\n", ((x)*(x)));

再使用：

SQR(8);

则输出的是：

The square of　8　is 64.

很简单吧？相信你现在已经明白#号的使用方法了。

3.8，##预算符

和#运算符一样，##运算符可以用于宏函数的替换部分。这个运算符把两个语言符号组

合成单个语言符号。看例子：

```
#define    XNAME(n)    x ## n
```

如果这样使用宏：

```
XNAME(8)
```

则会被展开成这样：

```
x8
```

看明白了没？##就是个粘合剂，将前后两部分粘合起来。

第四章 指针和数组

几乎每次讲课讲到指针和数组时，我总会反复不停的问学生：到底什么是指针？什么是数组？他们之间到底是什么样的关系。从几乎没人能回答明白到几乎都能回答明白，需要经历一段“惨绝人寰”的痛。指针是 C/C++的精华，如果未能很好地掌握指针，那 C/C++也基本等于没学。可惜，对于刚毕业的计算机系的学生，几乎没有人真正完全掌握了指针和数组、以及内存管理，甚至有的学生告诉我说：他们老师认为指针与数组太难，工作又少用，所以没有讲解。对于这样的学校与老师，我是彻底的无语。我没有资格去谴责或是鄙视谁，只是窃以为，这个老师肯怕自己都未掌握指针。大学里很多老师并未真正写过多少代码，不掌握指针的老师肯定存在，这样的老师教出来的学生如何能找到工作？而目前市面上的书对指针和数组的区别也是几乎避而不谈，这就更加加深了学生掌握的难度。我平时上课总是非常细致而又小心的向学生讲解这些知识，生怕一不小心就讲错或是误导了学生。还好，至少到目前为止，我教过的学生几乎都能掌握指针和数组及内存管理的要点，当然要到能运用自如的程度还远远不够，这需要大量的写代码才能达到。另外需要说明的是，讲课时为了让学生深刻的掌握这些知识，我举了很多各式各样的例子来帮助学生理解。所以，我也希望读者朋友能好好体味这些例子。

三个问题：

A)，什么是指针？

B)，什么是数组？

C)，数组和指针之间有什么样的关系？

4.1，指针

4.1.1，指针的内存布局

先看下面的例子：

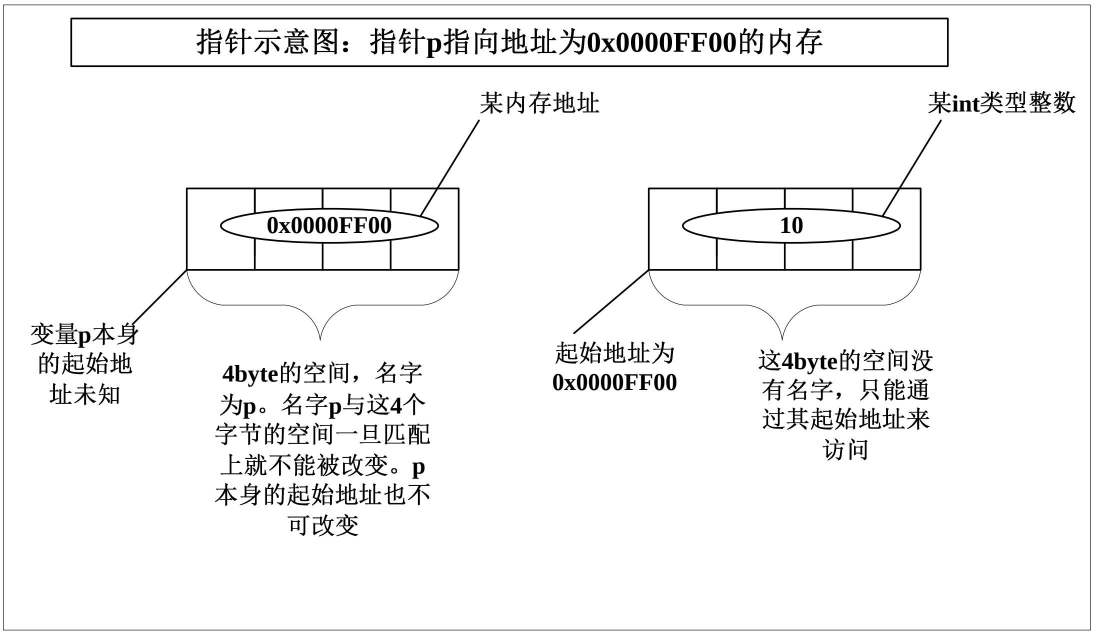
```
int *p;
```

大家都知道这里定义了一个指针 p。但是 p 到底是什么东西呢？还记得第一章里说过，“任何一种数据类型我们都可以把它当一个模子”吗？p，毫无疑问，是某个模子咔出来的。我们也讨论过，任何模子都必须有其特定的大小，这样才能用来“咔咔咔”。那咔出 p 的这个模子到底是什么样子呢？它占多大的空间呢？现在用 sizeof 测试一下（32 位系统）：sizeof（p）的值为 4。嗯，这说明咔出 p 的这个模子大小为 4 个 byte。显然，这个模子不是“int”，虽然它大小也为 4。既然不是“int”那就一定是“int *”了。好，那现在我们可以这么理解这个定义：

一个“int *”类型的模子在内存上咔出了 4 个字节的空间，然后把这个 4 个字节大小的

空间命名为 p，同时限定这 4 个字节的空间里面只能存储某个内存地址，即使你存入别的任何数据，都将被当作地址处理，而且这个内存地址开始的连续 4 个字节上只能存储某个 int 类型的数据。

这是一段咬文嚼字的说明，我们还是用图来解析一下：



如上图所示，我们把 p 称为指针变量,p 里存储的内存地址处的内存称为 p 所指向的内存。指针变量 p 里存储的任何数据都将被当作地址来处理。

我们可以简单的这么理解：一个基本的数据类型（包括结构体等自定义类型）加上“*”号就构成了一个指针类型的模子。这个模子的大小是一定的，与“*”号前面的数据类型无关。“*”号前面的数据类型只是说明指针所指向的内存里存储的数据类型。所以，在 32 位系统下，不管什么样的指针类型，其大小都为 4byte。可以测试一下 sizeof（void *）。

4.1.2，“*”与防盗门的钥匙

这里这个“*”号怎么理解呢？举个例子：当你回到家门口时，你想进屋第一件事就是拿出钥匙来开锁。那你想想防盗门的锁芯是不是很像这个“*”号？你要进屋必须要用钥匙，那你去读写一块内存是不是也要一把钥匙呢？这个“*”号就是不是就是我们最好的钥匙？使用指针的时候，没有它，你是不可能读写某块内存的。

4.1.3，int *p = NULL 和*p = NULL 有什么区别？

很多初学者都无法分清这两者之间的区别。我们先看下面的代码：

```
int *p = NULL;
```

这时候我们可以通过编译器查看 p 的值为 0x00000000。这句代码的意思是：定义一个指针变量 p，其指向的内存里面保存的是 int 类型的数据；在定义变量 p 的同时把 p 的值设置为 0x00000000，而不是把*p 的值设置为 0x00000000。这个过程叫做初始化，是在编译的时候进行的。

明白了什么是初始化之后，再看下面的代码：

```
int *p;
*p = NULL;
```

同样，我们可以在编译器上调试这两行代码。第一行代码，定义了一个指针变量 p，其指向的内存里面保存的是 int 类型的数据；但是这时候变量 p 本身的值是多少不得而知，也就是说现在变量 p 保存的有可能是一个非法的地址。第二行代码，给*p 赋值为 NULL，即给 p 指向的内存赋值为 NULL；但是由于 p 指向的内存可能是非法的，所以调试的时候编译器可能会报告一个内存访问错误。这样的话，我们可以把上面的代码改写改写，使 p 指向一块合法的内存：

```
int i = 10;
int *p = &i;
*p = NULL;
```

在编译器上调试一下，我们发现 p 指向的内存由原来的 10 变为 0 了；而 p 本身的值， 即内存地址并没有改变。

经过上面的分析，相信你已经明白它们之间的区别了。不过这里还有一个问题需要注意，也就是这个 NULL。初学者往往在这里犯错误。

注意 NULL 就是 NULL，它被宏定义为 0：

```
#define NULL 0
```

很多系统下除了有 NULL 外，还有 NUL（Visual C++ 6.0 上提示说不认识 NUL）。NUL 是 ASCII 码表的第一个字符，表示的是空字符，其 ASCII 码值为 0。其值虽然都为 0，但表示的意思完全不一样。同样，NULL 和 0 表示的意思也完全不一样。一定不要混淆。

另外还有初学者在使用 NULL 的时候误写成 null 或 Null 等。这些都是不正确的，C 语言对大小写十分敏感啊。当然，也确实有系统也定义了 null，其意思也与 NULL 没有区别，但是你千万不用使用 null，这会影响你代码的移植性。

4.1.4，如何将数值存储到指定的内存地址

假设现在需要往内存 0x12ff7c 地址上存入一个整型数 0x100。我们怎么才能做到呢？我们知道可以通过一个指针向其指向的内存地址写入数据，那么这里的内存地址 0x12ff7c 其本质不就是一个指针嘛。所以我们可以用下面的方法：

```
int *p = (int *)0x12ff7c;
*p = 0x100;
```

需要注意的是将地址 0x12ff7c 赋值给指针变量 p 的时候必须强制转换。至于这里为什么选择内存地址 0x12ff7c，而不选择别的地址，比如 0xff00 等。这仅仅是为了方便在 Visual C++ 6.0 上测试而已。如果你选择 0xff00，也许在执行*p = 0x100;这条语句的时候，编译器会报告一个内存访问的错误，因为地址 0xff00 处的内存你可能并没有权力去访问。既然这样，我们怎么知道一个内存地址是可以合法的被访问呢？也就是说你怎么知道地址 0x12ff7c 处的内存是可以被访问的呢？其实这很简单，我们可以先定义一个变量 i，比如：

```
int i = 0;
```

变量 i 所处的内存肯定是可以被访问的。然后在编译器的 watch 窗口上观察&i 的值不就知道其内存地址了么？这里我得到的地址是 0x12ff7c，仅此而已（不同的编译器可能每次给变量 i 分配的内存地址不一样，而刚好 Visual C++ 6.0 每次都一样）。你完全可以给任意一个可以被合法访问的地址赋值。得到这个地址后再把“int i = 0;”这句代码删除。一切“罪证”

销毁得一干二净，简直是做得天衣无缝。

除了这样就没有别的办法了吗？未必。我们甚至可以直接这么写代码：

```
*(int *)0x12ff7c = 0x100;
```

这行代码其实和上面的两行代码没有本质的区别。先将地址 0x12ff7c 强制转换，告诉编译器这个地址上将存储一个 int 类型的数据；然后通过钥匙"*"向这块内存写入一个数据。

上面讨论了这么多，其实其表达形式并不重要，重要的是这种思维方式。也就是说我们完全有办法给指定的某个内存地址写入数据的。

4.1.5，编译器的 bug？

另外一个有意思的现象，在 Visual C++ 6.0 调试如下代码的时候却又发现一个古怪的问题：

```
int *p = (int *)0x12ff7c;
*p = NULL;
p = NULL;
```

在执行完第二条代码之后，发现 p 的值变为 0x00000000 了。按照我么上一节的解释，应该p 的值不变，只是 p 指向的内存被赋值为 0。难道我们讲错了吗？别急，再试试如下代码：

```
int i = 10;
int *p = (int *)0x12ff7c;
*p = NULL;
p = NULL;
```

通过调试，发现这样子的话，p 的值没有变，而 p 指向的内存的值变为 0 了。这与我们前面讲解的完全一致。当然这里的 i 的地址刚好是 0x12ff7c，但这并不能改变"*p = NULL;"这行代码的功能。

为了再次测试这个问题，我又调试了如下代码：

```
int i = 10;
int j = 100;
int *p = (int *)0x12ff78;
*p = NULL;
p = NULL;
```

这里 0x12ff78 刚好就是变量 j 的地址。这样的话一切正常，但是如果把"int j = 100;"这行代码删除的话，又出现上述的问题了。测试到这里我还是不甘心，编译器怎么能犯这种低级错误呢？于是又接着进行了如下测试：

```
unsigned int i = 10;
//unsigned int j = 100;
unsigned int *p = (unsigned int *)0x12ff78;
*p = NULL;
p = NULL;
```

得到的结果与上面完全一样。当然，我还是没有死心，又进行了如下测试：

```
char    ch = 10;
char *p = (char *)0x12ff7c;
*p = NULL;
p = NULL;
```

这样子的话，完全正常。但当我删除掉第一行代码后再测试，这里的 p 的值并未变成 0x00000000，而是变成了 0x0012ff00，同时*p 的值变成了 0。这又是怎么回事呢？初学者是否认为这是编译器“良心发现”，把*p 的值改写为 0 了。

如果你真这么认为，那就大错特错了。这里的*p 还是地址 0x12ff7c 上的内容吗？显然不是，而是地址 0x0012ff00 上的内容。至于 0x12ff7c 为什么变成 0x0012ff00，则是因为编译器认为这是把 NULL 赋值给 char 类型的内存，所以只是把指针变量 p 的低地址上的一个字节赋值为 0。至于为什么是低地址，请参看前面讲解过大小端模式相关内容。

测试到这里，已经基本可以肯定这是 Visual C++ 6.0 的一个 bug。所以平时一定不要迷信某个编译器，要相信自己的判断。当然，后面还会提到一个我认为的 Visual C++ 6.0 的一个 bug。还有，这个小小的例子，你是否可以在多个编译器上测试测试呢？

4.1.6，如何达到手中无剑、胸中也无剑的地步

噢，上面的讨论一不小心就这么多了。这里我为什么要把这个小小的问题放到这里长篇大论呢？我是想告诉读者：研究问题一定要肯钻研。千万不要小看某一个简单的事情，简单的事情可能富含着很多秘密。经过这样一番深究，相信你也有不少收获。平时学习工作也是如此，不要小瞧任何一件简单的事情，把简单的事情做好也是一种伟大。劳模许振超开了几十年的吊车，技术精到指哪打哪的地步。达到这种程度是需要花苦功夫的，几十年如一日天天重复这件看似很简单的事情，这不是一般人能做到的。同样的，在《天龙八部》中，萧峰血战聚贤庄的时候，一套平平凡凡的太祖长拳打得虎虎生威，在场的英雄无不佩服至极，这也是其苦练的结果。我们学习工作同样如此，要肯下苦功夫钻研，不要怕钻得深，只怕钻得不深。其实这也就是为什么同一个班的学生，水平会相差非常大的最关键之处。学得好的，往往是那些舍得钻研的学生。我平时上课教学生的绝不仅仅是知识点，更多的时候我在教他们学习和解决问题的方法。有时候这个过程远比结论要重要的多。后面的内容，你也应该能看出来，我非常注重过程的分析，只有你真正明白了这些思考问题、解决问题的方法和过程，你才能真正立于不败之地。所有的问题对你来说都是一个样，没有本质的区别。解决任何问题的办法都一致，那就是把没见过的、不会的问题想法设法转换成你见过的、你会的问题；至于怎么去转换那就要靠你的苦学苦练了。也就是说你要达到手中无剑，胸中也无剑的地步。

当然这些只是我个人的领悟，写在这里希望能与君共勉。

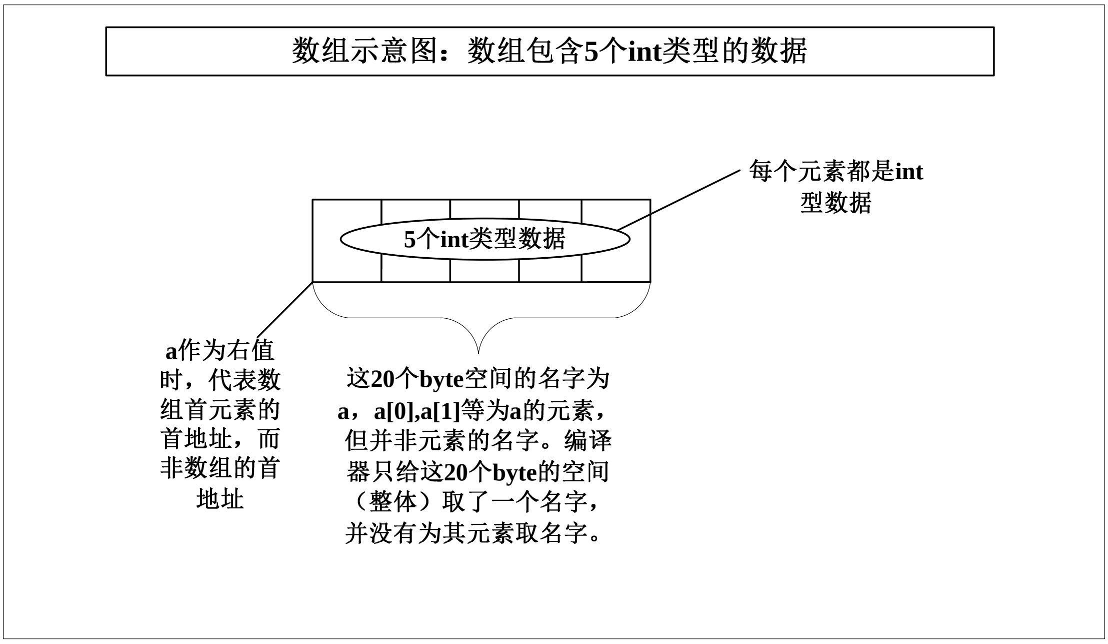
4.2，数组

4.2.1，数组的内存布局

先看下面的例子：

```
int a[5];
```

所有人都明白这里定义了一个数组，其包含了 5 个 int 型的数据。我们可以用 a[0],a[1]等来访问数组里面的每一个元素，那么这些元素的名字就是 a[0],a[1]…吗？看下面的示意图：



如上图所示，当我们定义一个数组 a 时，编译器根据指定的元素个数和元素的类型分配确定大小（元素类型大小*元素个数）的一块内存，并把这块内存的名字命名为 a。名字 a 一旦与这块内存匹配就不能被改变。a[0],a[1]等为 a 的元素，但并非元素的名字。数组的每一个元素都是没有名字的。那现在再来回答第一章讲解 sizeof 关键字时的几个问题：

sizeof(a)的值为 sizeof(int)*5，32 位系统下为 20。

sizeof(a[0])的值为 sizeof(int)，32 位系统下为 4。

sizeof(a[5])的值在 32 位系统下为 4。并没有出错，为什么呢？我们讲过 sizeof 是关键字不是函数。函数求值是在运行的时候，而**关键字 sizeof 求值是在编译**的时候。虽然并不存在 a[5]这个元素，但是这里也并没有去真正访问 a[5],而是仅仅根据数组元素的类型来确定其值。所以这里使用 a[5]并不会出错。

sizeof(&a[0])的值在 32 位系下为 4，这很好理解。取元素 a[0]的首地址。

sizeof(&a)的值在 32 位系统下也为 4，这也很好理解。取数组 a 的首地址。但是在 Visual C++6.0 上，这个值为 20，我认为是错误的。

4.2.2，省政府和市政的区别----&a[0]和&a 的区别

这里&a[0]和&a 到底有什么区别呢？a[0]是一个元素，a 是整个数组，虽然&a[0]和&a 的值一样，但其意义不一样。前者是数组首元素的首地址，而后者是数组的首地址。举个例子：湖南的省政府在长沙，而长沙的市政府也在长沙。两个政府都在长沙，但其代表的意义完全不同。这里也是同一个意思。

4.2.3，数组名 a 作为左值和右值的区别

简单而言，出现在赋值符“=”右边的就是右值，出现在赋值符“=”左边的就是左值。比如,x=y。

左值：在这个上下文环境中，编译器认为 x 的含义是 x 所代表的地址。这个地址只有编译器知道，在编译的时候确定，编译器在一个特定的区域保存这个地址，我们完全不必

考虑这个地址保存在哪里。

右值：在这个上下文环境中，编译器认为 y 的含义是 y 所代表的地址里面的内容。这个内容是什么，只有到运行时才知道。

C 语言引入一个术语-----“可修改的左值”。意思就是，出现在赋值符左边的符号所代表的地址上的内容一定是可以被修改的。换句话说，就是我们只能给非只读变量赋值。

既然已经明白左值和右值的区别，下面就讨论一下数组作为左值和右值的情况：

当 a 作为右值的时候代表的是什么意思呢？很多书认为是数组的首地址，其实这是非常错误的。a 作为右值时其意义与&a[0]是一样，代表的是**数组首元素的首地址**，而不是数组的首地址。这是两码事。但是注意，这仅仅是代表，并没有一个地方（这只是简单的这么认为，其具体实现细节不作过多讨论）来存储这个地址，也就是说编译器并没有为数组 a 分配一块内存来存其地址，这一点就与指针有很大的差别。

a 作为右值，我们清楚了其含义，那作为左值呢？

a 不能作为左值！这个错误几乎每一个学生都犯过。编译器会认为数组名作为左值代表的意思是 a 的首元素的首地址，但是这个地址开始的一块内存是一个总体，我们只能访问数组的某个元素而无法把数组当一个总体进行访问。所以我们可以把 a[i]当左值，而无法把 a 当左值。其实我们完全可以把 a 当一个普通的变量来看，只不过这个变量内部分为很多小块，我们只能通过分别访问这些小块来达到访问整个变量 a 的目的。

4.3，指针与数组之间的恩恩怨怨

很多初学者弄不清指针和数组到底有什么样的关系。我现在就告诉你：他们之间没有任何关系！只是他们经常穿着相似的衣服来逗你玩罢了。

指针就是指针，指针变量在 32 位系统下，永远占 4 个 byte，其值为某一个内存的地址。指针可以指向任何地方，但是不是任何地方你都能通过这个指针变量访问到。

数组就是数组，其大小与元素的类型和个数有关。定义数组时必须指定其元素的类型和个数。数组可以存任何类型的数据，但不能存函数。

既然它们之间没有任何关系，那为何很多人把数组和指针混淆呢？甚至很多人认为指针和数组是一样的。这就与市面上的 C 语言的书有关，几乎没有一本书把这个问题讲透彻，讲明白了。

4.3.1，以指针的形式访问和以下标的形式访问

下面我们就详细讨论讨论它们之间似是而非的一些特点。例如，函数内部有如下定义：

A),char *p = “abcdef”;

B),char a[] = “123456”;

4.3.1.1，以指针的形式访问和以下标的形式访问指针

例子 A)定义了一个指针变量 p，p 本身在栈上占 4 个 byte，p 里存储的是一块内存的首地址。这块内存在静态区，其空间大小为 7 个 byte，这块内存也没有名字。对这块内存的访问完全是匿名的访问。比如现在需要读取字符‘e’，我们有两种方式：

1），以指针的形式：*(p+4)。先取出 p 里存储的地址值，假设为 0x0000FF00，然后加上 4 个字符的偏移量，得到新的地址 0x0000FF04。然后取出 0x0000FF04 地址上的值。

2），以下标的形式：p[4]。编译器总是把以下标的形式的操作解析为以指针的形式的操作。p[4]这个操作会被解析成：先取出 p 里存储的地址值，然后加上中括号中 4 个元素的偏移量，计算出新的地址，然后从新的地址中取出值。也就是说**以下标的形式访问在本质上与以指针的形式访问没有区别，只是写法上不同罢了。**

4.3.1.2，以指针的形式访问和以下标的形式访问数组

例子 B)定义了一个数组 a，a 拥有 7 个 char 类型的元素，其空间大小为 7。数组 a 本身在栈上面。对 a 的元素的访问必须先根据数组的名字 a 找到数组首元素的首地址，然后根据偏移量找到相应的值。这是一种典型的“具名+匿名”访问。比如现在需要读取字符‘5’，我们有两种方式：

1），以指针的形式：*(a+4)。a 这时候代表的是数组首元素的首地址，假设为 0x0000FF00，然后加上 4 个字符的偏移量，得到新的地址 0x0000FF04。然后取出 0x0000FF04 地址上的值。

2），以下标的形式：a[4]。编译器总是把以下标的形式的操作解析为以指针的形式的操作。a[4]这个操作会被解析成：a 作为数组首元素的首地址，然后加上中括号中 4 个元素的偏移量，计算出新的地址，然后从新的地址中取出值。

由上面的分析，我们可以看到，指针和数组根本就是两个完全不一样的东西。只是它们都可以“以指针形式”或“以下标形式”进行访问。一个是完全的匿名访问，一个是典型的具名+匿名访问。一定要注意的是这个“以 XXX 的形式的访问”这种表达方式。

另外一个需要强调的是：上面所说的偏移量 4 代表的是 4 个元素，而不是 4 个 byte。只不过这里刚好是 char 类型数据 1 个字符的大小就为 1 个 byte。记住这个**偏移量的单位是元素的个数**而不是 byte 数，在计算新地址时千万别弄错了。

4.3.2，a 和&a 的区别

通过上面的分析，相信你已经明白数组和指针的访问方式了，下面再看这个例子：

```
main()
{
    int a[5]={1,2,3,4,5};
    int *ptr=(int *)(&a+1);
    printf("%d,%d",*(a+1),*(ptr-1));
}
```

打印出来的值为多少呢？这里主要是考查关于指针加减操作的理解。

对指针进行加 1 操作，得到的是下一个元素的地址，而不是原有地址值直接加 1。所以，**一个类型为 T 的指针的移动，以 sizeof(T) 为移动单位**。 因此，对上题来说，a 是一个一维数组，数组中有 5 个元素； ptr 是一个 int 型的指针。

&a + 1: 取数组 a 的首地址，该地址的值加上 sizeof(a) 的值，即 &a + 5*sizeof(int)，也就是下一个数组的首地址，显然当前指针已经越过了数组的界限。

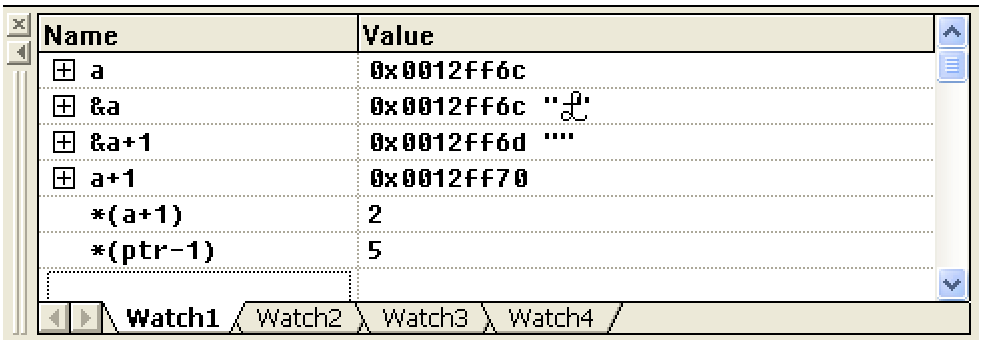
(int *)(&a+1): 则是把上一步计算出来的地址，强制转换为 int * 类型，赋值给 ptr。

*(a+1): a,&a 的值是一样的，但意思不一样，a 是数组首元素的首地址，也就是 a[0]的首地址，&a 是数组的首地址，a+1 是数组下一元素的首地址，即 a[1]的首地址,&a+1 是下一个数组的首地址。所以输出 2

*(ptr-1): 因为 ptr 是指向 a[5]，并且 ptr 是 int * 类型，所以 *(ptr-1) 是指向 a[4]，输出 5。

这些分析我相信大家都能理解，但是在授课时，学生向我提出了如下问题：

在 Visual C++6.0 的 Watch 窗口中&a+1 的值怎么会是（x0012ff6d（0x0012ff6c+1）呢？



上图是在 Visual C++6.0 调试本函数时的截图。

a 在这里代表是的数组首元素的地址即 a[0]的首地址，其值为 0x0012ff6c。

&a 代表的是数组的首地址，其值为 0x0012ff6c。

a+1 的值是 0x0012ff6c+1*sizeof（int），等于 0x0012ff70。

问题就是&a+1 的值怎么会是（x0012ff6d（0x0012ff6c+1）呢？

按照我们上面的分析应该为 0x0012ff6c+5*sizeof（int）。其实很好理解。当你把&a+1 放到 Watch 窗口中观察其值时，表达式&a+1 已经脱离其上下文环境，编译器就很简单的把它解析为&a 的值然后加上 1byte。而 a+1 的解析就正确，我认为这是 Visual C++6.0 的一个 bug。既然如此，我们怎么证明证明&a+1 的值确实为 0x0012ff6c+5*sizeof（int）呢？很好办，用 printf 函数打印出来。这就是我在本书前言里所说的，有的时候我们确实需要 printf 函数才能解决问题。你可以试试用 printf("%x",&a+1);打印其值，看是否为 0x0012ff6c+5*sizeof（int）。注意如果你用的是 printf("%d",&a+1);打印，那你必须在十进制和十六进制之间换算一下，不要冤枉了编译器。

另外我要强调一点：不到非不得已，尽量别使用 printf 函数，它会使你养成只看结果不问为什么的习惯。比如这个列子，*(a+1)和*(ptr-1)的值完全可以通过 Watch 窗口来查看。

平时初学者很喜欢用“printf("%d,%d",*(a+1),*(ptr-1));”这类的表达式来直接打印出值，如果发现值是正确的就欢天喜地。这个时候往往认为自己的代码没有问题，根本就不去查

看其变量的值，更别说是内存和寄存器的值了。更有甚者，printf 函数打印出来的值不正确，就措手无策，举手问“老师，我这里为什么不对啊？”。长此以往就养成了很不好的习惯，只看结果，不重调试。这就是为什么同样的几年经验，有的人水平很高，而有的人水平却很低。其根本原因就在于此，往往被一些表面现象所迷惑。printf 函数打印出来的值是对的就能说明你的代码一定没问题吗？我看未必。曾经一个学生，我让其实现直接插入排序算法。很快他把函数写完了，把值用 printf 函数打印出来给我看。我看其代码却发现他使用的算法本质上其实是冒泡排序，只是写得像直接插入排序罢了。等等这种情况数都数不过来，往往犯了错误还以为自己是对的。所以我平时上课之前往往会强调，不到非不得已，不允许使用 printf 函数，而要自己去查看变量和内存的值。学生的这种不好的习惯也与目前市面上的教材、参考书有关，这些书甚至花大篇幅来介绍 scanf 和 printf 这类的函数，却几乎不讲解调试技术。甚至有的书还在讲 TruboC 2.0 之类的调试器！如此教材教出来的学生质量可想而知。

4.3.3，指针和数组的定义与声明

4.3.3.1，定义为数组，声明为指针

文件 1 中定义如下：

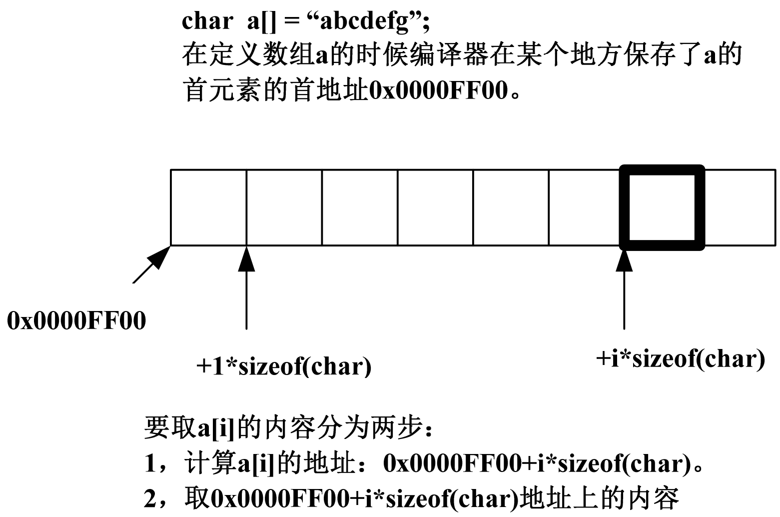
```
char a[100];
```

文件 2 中声明如下（关于 extern 的用法，以及定义和声明的区别，请复习第一章）：

```
extern char *a;
```

这里，文件 1 中定义了数组 a，文件 2 中声明它为指针。这有什么问题吗？平时不是总说数组与指针相似，甚至可以通用吗？但是，很不幸，这是错误的。通过上面的分析我们也能明白一些，但是“革命尚未成功，同志仍需努力”。你或许还记得我上面说过的话：**数组就是数组，指针就是指针，它们是完全不同的两码事！他们之间没有任何关系，只是经常穿着相似的衣服来迷惑你罢了。**下面就来分析分析这个问题：

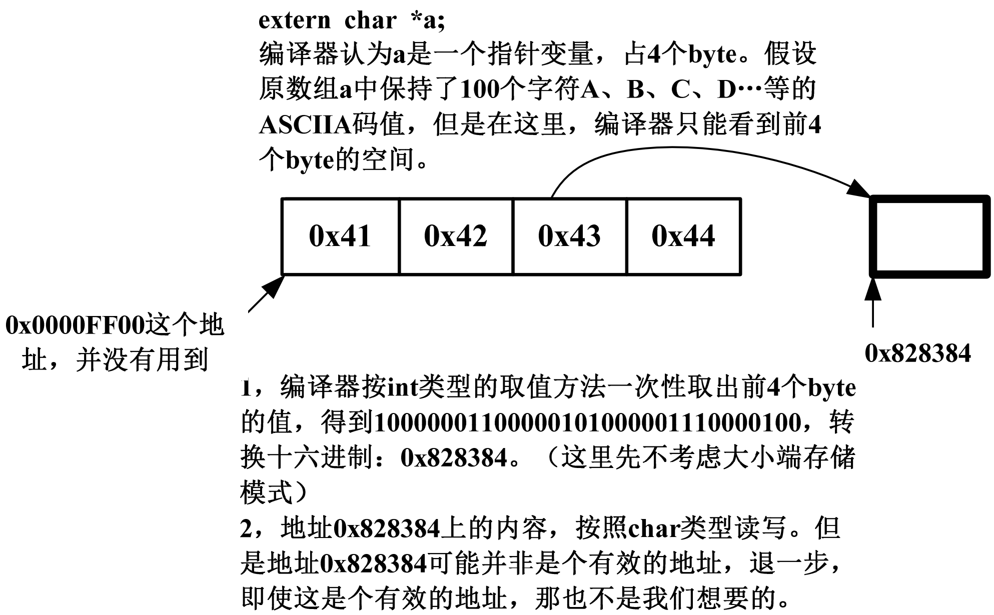
在第一章的开始，我就强调了定义和声明之间的区别，定义分配的内存，而声明没有。定义只能出现一次，而声明可以出现多次。这里 extern 告诉编译器 a 这个名字已经在别的文件中被定义了，下面的代码使用的名字 a 是别的文件定义的。再回顾到前面对于左值和右值的讨论，我们知道如果编译器需要某个地址（可能还需要加上偏移量）来执行某种操作的话，它就可以直接通过开锁动作(使用“*”这把钥匙)来读或者写这个地址上的内存，并不需要先去找到储存这个地址的地方。相反，对于指针而言，必须先去找到储存这个地址的地方，取出这个地址值然后对这个地址进行开锁（使用“*”这把钥匙）。如下图：



这就是为什么 extern char a[]与 extern char a[100]等价的原因。因为这只是声明，不分配空间，所以编译器无需知道这个数组有多少个元素。这两个声明都告诉编译器 a 是在别的文件中被定义的一个数组，a 同时代表着数组 a 的首元素的首地址，也就是这块内存的起始地址。数组内地任何元素的的地址都只需要知道这个地址就可以计算出来。

但是，当你声明为 extern char *a 时，编译器理所当然的认为 a 是一个指针变量，在 32 位系统下，占 4 个 byte。这 4 个 byte 里保存了一个地址，这个地址上存的是字符类型数据。虽然在文件 1 中，编译器知道 a 是一个数组，但是在文件 2 中，编译器并不知道这点。大多数编译器是按文件分别编译的，编译器只按照本文件中声明的类型来处理。所以，虽然 a 实际大小为 100 个 byte，但是在文件 2 中，编译器认为 a 只占 4 个 byte。

我们说过，编译器会把存在指针变量中的任何数据当作地址来处理。所以，如果需要访问这些字符类型数据，我们必须先从指针变量 a 中取出其保存的地址。如下图：



4.3.3.2，定义为指针，声明为数组

显然，按照上面的分析，我们把文件 1 中定义的数组在文件 2 中声明为指针会发生错误。同样的，如果在文件 1 中定义为指针，而在文件中声明为数组也会发生错误：

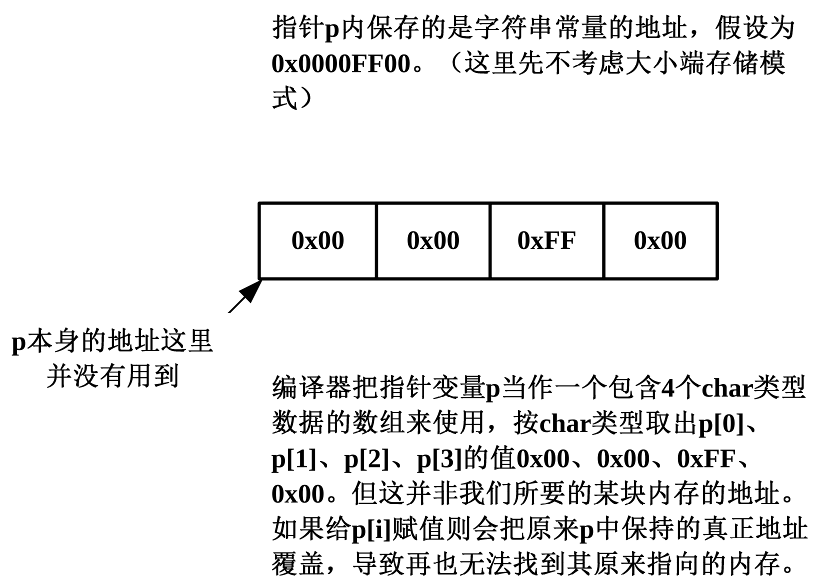
文件 1

```
char *p = “abcdefg”;
```

文件 2

```
extern char p[];
```

在文件 1 中，编译器分配 4 个 byte 空间，并命名为 p。同时 p 里保存了字符串常量“abcdefg”的首字符的首地址。这个字符串常量本身保存在内存的静态区，其内容不可更改。在文件 2 中，编译器认为 p 是一个数组，其大小为 4 个 byte，数组内保存的是 char 类型的数据。在文件 2 中使用 p 的过程如下图：



4.3.4，指针和数组的对比

通过上面的分析，相信你已经知道数组与指针的的确确是两码事了。他们之间是不可以混淆的，但是我们可以“以 XXXX 的形式”访问数组的元素或指针指向的内容。以后一定要确认你的代码在一个地方定义为指针，在别的地方也只能声明为指针；在一个的地方定义为数组，在别的地方也只能声明为数组。切记不可混淆。下面再用一个表来总结一下指针和数组的特性：

指针	数组
保存数据的地址，任何存入指针变量 p 的数据都会被当作地址来处理。p 本身的地址由编译器另外存储，存储在哪里，我们并不知	保存数据，数组名 a 代表的是数组首元素的首地址而不是数组的首地址。&a 才是整个数组的首地址。a 本身的地址由编译器另外存

道。	储，存储在哪里，我们并不知道。
间接访问数据，首先取得指针变量 p 的内容，把它作为地址，然后从这个地址提取数据或向这个地址写入数据。指针可以以指针的形式访问*(p+i); 也可以以下标的形式访问 p[i]。但其本质都是先取 p 的内容然后加上 i*sizeof(类型)个 byte 作为数据的真正地址。	直接访问数据，数组名 a 是整个数组的名字，数组内每个元素并没有名字。只能通过“具名+匿名”的方式来访问其某个元素，不能把数组当一个整体来进行读写操作。数组可以以指针的形式访问*(a+i); 也可以以下标的形式访问 a[i]。但其本质都是 a 所代表的数组首元素的首地址加上 i*sizeof(类型)个 byte 作为数据的真正地址。
通常用于动态数据结构	通常用于存储固定数目且数据类型相同的元素。
相关的函数为 malloc 和 free。	隐式分配和删除
通常指向匿名数据（当然也可指向具名数据）	自身即为数组名

4.4，指针数组和数组指针

4.4.1，指针数组和数组指针的内存布局

初学者总是分不出指针数组与数组指针的区别。其实很好理解：

指针数组：首先它是一个数组，数组的元素都是指针，数组占多少个字节由数组本身决定。它是“储存指针的数组”的简称。

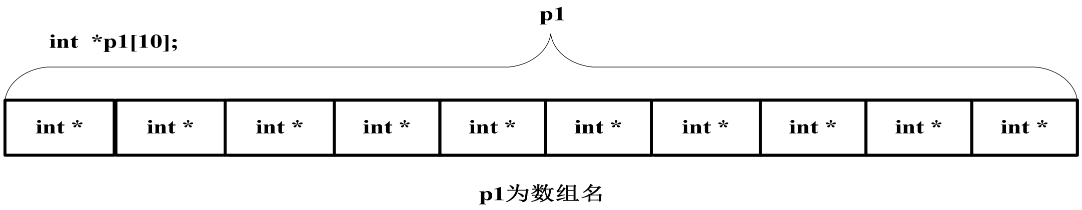
数组指针：首先它是一个指针，它指向一个数组。在 32 位系统下永远是占 4 个字节，至于它指向的数组占多少字节，不知道。它是“指向数组的指针”的简称。

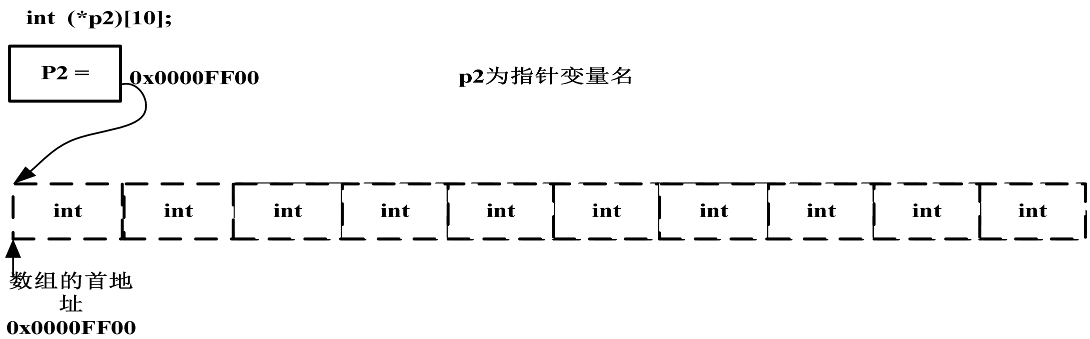
下面到底哪个是数组指针，哪个是指针数组呢：

A)，int *p1[10];

B)，int (*p2)[10];

每次上课问这个问题，总有弄不清楚的。这里需要明白一个符号之间的优先级问题。“[]”的优先级比“*”要高。p1 先与“[]”结合，构成一个数组的定义，数组名为 p1，int * 修饰的是数组的内容，即数组的每个元素。那现在我们清楚，这是一个数组，其包含 10 个指向 int 类型数据的指针，即指针数组。至于 p2 就更好理解了，在这里“()”的优先级比“[]”高，“*”号和 p2 构成一个指针的定义，指针变量名为 p2，int 修饰的是数组的内容，即数组的每个元素。数组在这里并没有名字，是个匿名数组。那现在我们清楚 p2 是一个指针，它指向一个包含 10 个 int 类型数据的数组，即数组指针。我们可以借助下面的图加深理解：





4.4.2，int　(*)[10]　p2-----也许应该这么定义数组指针

这里有个有意思的话题值得探讨一下：平时我们定义指针不都是在数据类型后面加上指针变量名么？这个指针 p2 的定义怎么不是按照这个语法来定义的呢？也许我们应该这样来定义 p2：

int　(*)[10]　p2;

int　(*)[10]是指针类型，p2 是指针变量。这样看起来的确不错，不过就是样子有些别扭。其实数组指针的原型确实就是这样子的，只不过为了方便与好看把指针变量 p2 前移了而已。你私下完全可以这么理解这点。虽然编译器不这么想。^_^

4.4.3，再论 a 和&a 之间的区别

既然这样，那问题就来了。前面我们讲过 a 和&a 之间的区别，现在再来看看下面的代码：

```
int   main()
{
    char a[5]={'A','B','C','D'};
    char (*p3)[5] = &a;
    char (*p4)[5] = a;
    return 0;
}
```

上面对 p3 和 p4 的使用，哪个正确呢？p3+1 的值会是什么？p4+1 的值又会是什么？

毫无疑问，p3 和 p4 都是数组指针，指向的是整个数组。&a 是整个数组的首地址，a 是数组首元素的首地址，其值相同但意义不同。在 C 语言里，赋值符号“=”号两边的数据类型必须是相同的，如果不同需要显示或隐式的类型转换。p3 这个定义的“=”号两边的数据类型完全一致，而 p4 这个定义的“=”号两边的数据类型就不一致了。左边的类型是指向整个数组的指针，右边的数据类型是指向单个字符的指针。在 Visual C++6.0 上给出如下警告：warning C4047: 'initializing' : 'char (*)[5]' differs in levels of indirection from 'char *'。还好，这里虽然给出了警告，但由于&a 和 a 的值一样，而变量作为右值时编译器只是取变量的值，所以运行并没有什么问题。不过我仍然警告你别这么用。

既然现在清楚了 p3 和 p4 都是指向整个数组的，那 p3+1 和 p4+1 的值就很好理解了。但是如果修改一下代码，会有什么问题？p3+1 和 p4+1 的值又是多少呢？

```
int    main()
{
       char a[5]={'A','B','C','D'};
       char (*p3)[3] = &a;
       char (*p4)[3] = a;
       return 0;
}
```

甚至还可以把代码再修改：

```
int    main()
{
       char a[5]={'A','B','C','D'};
       char (*p3)[10] = &a;
       char (*p4)[10] = a;
       return 0;
}
```

这个时候又会有什么样的问题？p3+1 和 p4+1 的值又是多少？

上述几个问题，希望读者能仔细考虑考虑。

4.4.4，地址的强制转换

先看下面这个例子：

```
struct Test
{
    int        Num;
    char    *pcName;
```

```
        short    sDate;
        char    cha[2];
        short    sBa[4];
    }*p;
```

假设 p 的值为 0x100000。 如下表表达式的值分别为多少？

p + 0x1 = 0x___ ?

(unsigned long)p + 0x1 = 0x___?

(unsigned int*)p + 0x1 = 0x___?

我相信会有很多人一开始没看明白这个问题是什么意思。其实我们再仔细看看，这个知识点似曾相识。一个指针变量与一个整数相加减，到底该怎么解析呢？

还记得前面我们的表达式“a+1”与“&a+1”之间的区别吗？其实这里也一样。指针变量与一个整数相加减并不是用指针变量里的地址直接加减这个整数。这个整数的单位不是 byte 而是元素的个数。所以：

p + 0x1 的值为 0x100000+sizof（Test）*0x1。至于此结构体的大小为 20byte，前面的章节已经详细讲解过。所以 p + 0x1 的值为：0x100014。

(unsigned long)p + 0x1 的值呢？这里涉及到强制转换，将指针变量 p 保存的值强制转换成无符号的长整型数。任何数值一旦被强制转换，其类型就改变了。所以这个表达式其实就是一个无符号的长整型数加上另一个整数。所以其值为：0x100001。

(unsigned int*)p + 0x1 的值呢？这里的 p 被强制转换成一个指向无符号整型的指针。所以其值为：0x100000+sizof（unsigned int）*0x1，等于 0x100004。

上面这个问题似乎还没啥技术含量，下面就来个有技术含量的：

在 x86 系统下，其值为多少？

```
int main()
{
    int a[4]={1,2,3,4};
    int *ptr1=(int *)(&a+1);
    int *ptr2=(int *)((int)a+1);

    printf("%x,%x",ptr1[-1],*ptr2);

    return 0;
}
```

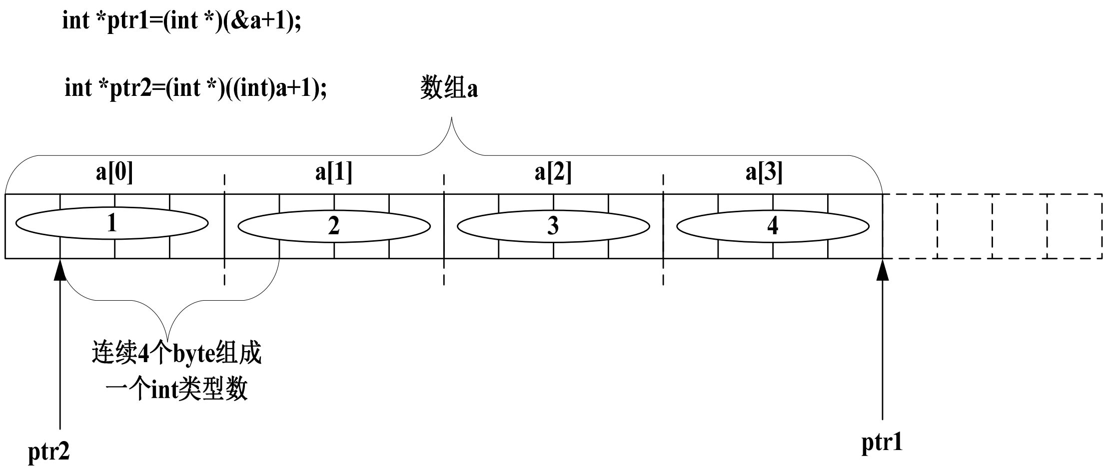
这是我讲课时一个学生问我的题，他在网上看到的，据说难倒了 n 个人。我看题之后告诉他，这些人肯定不懂汇编，一个懂汇编的人，这种题实在是小 case。下面就来分析分析这个问题：

根据上面的讲解，&a+1 与 a+1 的区别已经清楚。

ptr1：将&a+1 的值强制转换成 int*类型，赋值给 int* 类型的变量 ptr，ptr1 肯定指到数组 a 的下一个 int 类型数据了。ptr1[-1]被解析成*(ptr1-1)，即 ptr1 往后退 4 个 byte。所以其值为 0x4。

ptr2：按照上面的讲解，(int)a+1 的值是元素 a[0]的第二个字节的地址。然后把这个地址强制转换成 int*类型的值赋给 ptr2，也就是说*ptr2 的值应该为元素 a[0]的第二个字节开始的连续 4 个 byte 的内容。

其内存布局如下图：



好，问题就来了，这连续 4 个 byte 里到底存了什么东西呢？也就是说元素 a[0],a[1]里面的值到底怎么存储的。这就涉及到系统的大小端模式了，如果懂汇编的话，这根本就不是问题。既然不知道当前系统是什么模式，那就得想办法测试。大小端模式与测试的方法在第一章讲解 union 关键字时已经详细讨论过了，请翻到彼处参看，这里就不再详述。我们可以用下面这个函数来测试当前系统的模式。

```
int checkSystem( )
{
     union check
     {
          int i;
          char ch;
     } c;
    c.i = 1;
    return (c.ch ==1);
}
```

如果当前系统为大端模式这个函数返回 0；如果为小端模式，函数返回 1。

也就是说如果此函数的返回值为 1 的话，*ptr2 的值为 0x2000000。

如果此函数的返回值为 0 的话，*ptr2 的值为 0x100。

4.5，多维数组与多级指针

多维数组与多级指针也是初学者感觉迷糊的一个地方。超过二维的数组和超过二级的指针其实并不多用。如果能弄明白二维数组与二级指针，那二维以上的也不是什么问题了。所以本节重点讨论二维数组与二级指针。

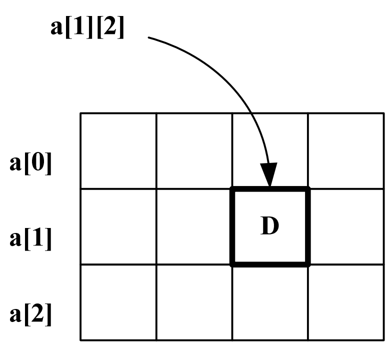
4.5.1，二维数组

4.5.1.1，假想中的二维数组布局

我们前面讨论过，数组里面可以存任何数据，除了函数。下面就详细讨论讨论数组里面存数组的情况。Excel 表，我相信大家都见过。我们平时就可以把二维数组假想成一个 excel 表，比如：

char a[3][4];

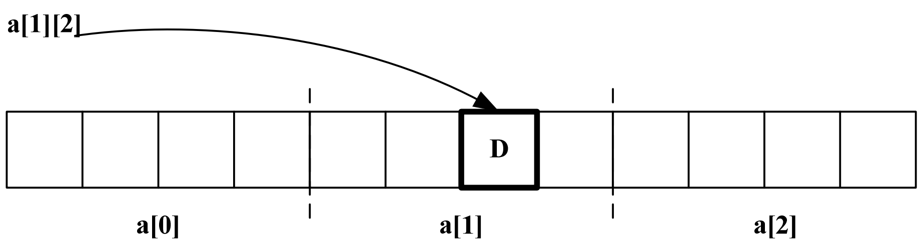
假想中的二维数组布局



4.5.1.2，内存与尺子的对比

实际上内存不是表状的，而是线性的。见过尺子吧？尺子和我们的内存非常相似。一般尺子上最小刻度为毫米，而内存的最小单位为 1 个 byte。平时我们说 32 毫米，是指以零开始偏移 32 毫米；平时我们说内存地址为 0x0000FF00 也是指从内存零地址开始偏移 0x0000FF00 个 byte。既然内存是线性的，那二维数组在内存里面肯定也是线性存储的。实际上其内存布局如下图：

现实中的二维数组布局



以数组下标的方式来访问其中的某个元素：a[i][j]。编译器总是将二维数组看成是一个一维数组，而一维数组的每一个元素又都是一个数组。a[3]这个一维数组的三个元素分别为：a[0],a[1],a[2]。每个元素的大小为 sizeof（a[0]）,即 sizof(char)*4。由此可以计算出 a[0],a[1],a[2]三个元素的首地址分别为& a[0]，& a[0]+ 1*sizof(char)*4，& a[0]+ 2*sizof(char)*4。亦即 a[i]的首地址为& a[0]+ i*sizof(char)*4。这时候再考虑 a[i]里面的内容。就本例而言，a[i]内有 4 个 char 类型的元素，其每个元素的首地址分别为&a[i]，&a[i]+1*sizof(char)，&a[i]+2*sizof(char)，&a[i]+3*sizof(char)，即 a[i][j]的首地址为&a[i]+j*sizof(char)。再把&a[i]

的值用 a 表示，得到 a[i][j]元素的首地址为：a+ i*sizof(char)*4+ j*sizof(char)。同样，可以换算成以指针的形式表示：*(*(a+i)+j)。

经过上面的讲解，相信你已经掌握了二维数组在内存里面的布局了。下面就看一个题：

```
#include <stdio.h>
int main(int argc,char * argv[])
{
      int a [3][2]={(0,1),(2,3),(4,5)};
      int *p;
      p=a [0];
      printf("%d",p[0]);
}
```

问打印出来的结果是多少？

很多人都觉得这太简单了，很快就能把答案告诉我：0。不过很可惜，错了。答案应该是 1。如果你也认为是 0，那你实在应该好好看看这个题。花括号里面嵌套的是小括号，而不是花括号！这里是花括号里面嵌套了逗号表达式！其实这个赋值就相当于 int a [3][2]={ 1, 3, 5};

所以，在初始化二维数组的时候一定要注意，别不小心把应该用的花括号写成小括号了。

4.5.1.3，&p[4][2] - &a[4][2]的值为多少？

上面的问题似乎还比较好理解，下面再看一个例子：

```
int    a[5][5];
int    (*p)[4];
p = a;
```

问&p[4][2] - &a[4][2]的值为多少？

这个问题似乎非常简单，但是几乎没有人答对了。我们可以先写代码测试一下其值，然后分析一下到底是为什么。在 Visual C++6.0 里，测试代码如下：

```
int main()
{
    int a[5][5];
    int (*p)[4];
    p = a;
    printf("a_ptr=%#p,p_ptr=%#p\n",&a[4][2],&p[4][2]);
    printf("%p,%d\n",&p[4][2] - &a[4][2],&p[4][2] - &a[4][2]);

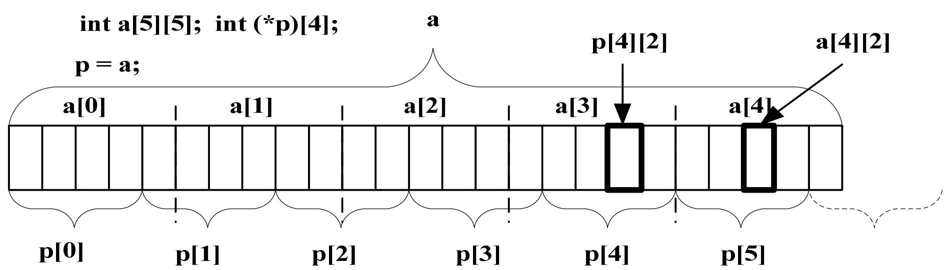

    return 0;
}
```

经过测试，可知&p[4][2] - &a[4][2]的值为-4。这到底是为什么呢？下面我们就来分析一下：

前面我们讲过，当数组名 a 作为右值时，代表的是数组首元素的首地址。这里的 a 为二维数组,我们把数组 a 看作是包含 5 个 int 类型元素的一维数组,里面再存储了一个一维数组。如此，则 a 在这里代表的是 a[0]的首地址。a+1 表示的是一维数组 a 的第二个元素。a[4]表示的是一维数组 a 的第 5 个元素，而这个元素里又存了一个一维数组。所以&a[4][2]表示的是&a[0][0]+4*5*sizeof(int) + 2*sizeof(int)。

根据定义，p 是指向一个包含 4 个元素的数组的指针。也就是说 p+1 表示的是指针 p 向后移动了一个“包含 4 个 int 类型元素的数组”。这里 1 的单位是 p 所指向的空间，即 4*sizeof(int)。所以，p[4]相对于 p[0]来说是向后移动了 4 个“包含 4 个 int 类型元素的数组”，即&p[4]表示的是&p[0]+4*4*sizeof(int)。由于 p 被初始化为&a[0]，那么&p[4][2]表示的是&a[0][0]+4*4*sizeof(int)+2* sizeof(int)。

再由上面的讲述，&p[4][2] 和 &a[4][2]的值相差 4 个 int 类型的元素。现在，上面测试出来的结果也可以理解了吧？其实我们最简单的办法就是画内存布局图：



这里最重要的一点就是明白数组指针 p 所指向的内存到底是什么。解决这类问题的最好办法就是画内存布局图。

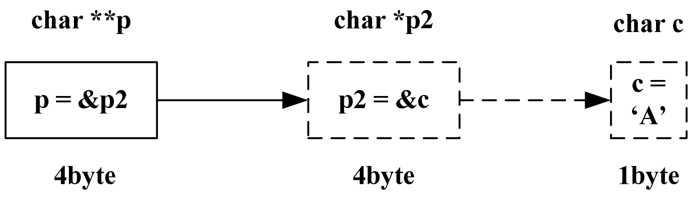
4.5.2，二级指针

4.5.2.1，二级指针的内存布局

二级指针是经常用到的，尤其与二维数组在一起的时候更是令人迷糊。例如：

char **p;

定义了一个二级指针变量 p。p 是一个指针变量，毫无疑问在 32 位系统下占 4 个 byte。它与一级指针不同的是，一级指针保存的是数据的地址，二级指针保存的是一级指针的地址。下图帮助理解：



我们试着给变量 p 初始化：

A)，p = NULL;

B)，char *p2; p = &p2;

任何指针变量都可以被初始化为 NULL（注意是 NULL，不是 NUL，更不是 null），二级指针也不例外。也就是说把指针指向数组的零地址。联想到前面我们把尺子比作内存，如果把内存初始化为 NULL，就相当于把指针指向尺子上 0 毫米处，这时候指针没有任何内存可用。

当我们真正需要使用 p 的时候，就必须把一个一级指针的地址保存到 p 中，所以 B）的赋值方式也是正确的。

给 p 赋值没有问题，但怎么使用 p 呢？这就需要我们前面多次提到的钥匙（“*”）。

第一步：根据 p 这个变量，取出它里面存的地址。

第二步：找到这个地址所在的内存。

第三步：用钥匙打开这块内存，取出它里面的地址，*p 的值。

第四步：找到第二次取出的这个地址。

第五步：用钥匙打开这块内存，取出它里面的内容，这就是我们真正的数据，**p 的值。

我们在这里用了两次钥匙（“*”）才最终取出了真正的数据。也就是说要取出二级指针所真正指向的数据，需要使用两次两次钥匙（“*”）。

至于超过二维的数组和超过二维的指针一般使用比较少，而且按照上面的分析方法同样也可以很轻松的分析明白，这里就不再详细讨论。读者有兴趣的话，可以研究研究。

4.6，数组参数与指针参数

我们都知道参数分为形参和实参。形参是指声明或定义函数时的参数，而实参是在调用函数时主调函数传递过来的实际值。

4.6.1，一维数组参数

4.6.1.1，能否向函数传递一个数组？

看例子：

```
void fun(char   a[10])
{
    char c = a[3];
}
```

```
int main()
{
    char b[10] =   "abcdefg";
    fun(b[10]);


    return 0;
}
```

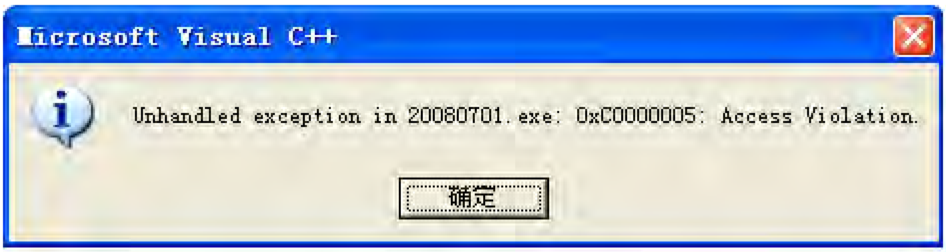
先看上面的调用，fun(b[10]);将 b[10]这个数组传递到 fun 函数。但这样正确吗？b[10]是代表一个数组吗？

显然不是，我们知道 b[0]代表是数组的一个元素，那 b[10]又何尝不是呢？只不过这里数组越界了，这个 b[10]并不存在。但在编译阶段，编译器并不会真正计算 b[10]的地址并取值，所以在编译的时候编译器并不认为这样有错误。虽然没有错误，但是编译器仍然给出了两个警告：

warning C4047: 'function' : 'char *' differs in levels of indirection from 'char '

warning C4024: 'fun' : different types for formal and actual parameter 1

这是什么意思呢？这两个警告告诉我们，函数参数需要的是一个 char*类型的参数，而实际参数为 char 类型，不匹配。虽然编译器没有给出错误，但是这样运行肯定会有问题。如图：



这是一个内存异常，我们分析分析其原因。其实这里至少有两个严重的错误。

第一：b[10]并不存在，在编译的时候由于没有去实际地址取值，所以没有出错，但是在运行时，将计算 b[10]的实际地址，并且取值。这时候发生越界错误。

第二：编译器的警告已经告诉我们编译器需要的是一个 char*类型的参数，而传递过去的是一个 char 类型的参数，这时候 fun 函数会将传入的 char 类型的数据当地址处理，同样会发生错误。（这点前面已经详细讲解）

第一个错误很好理解，那么第二个错误怎么理解呢？fun 函数明明传递的是一个数组啊，编译器怎么会说是 char *类型呢？别急，我们先把函数的调用方式改变一下：

fun(b);

b 是一个数组，现在将数组 b 作为实际参数传递。这下该没有问题了吧？调试、运行，一切正常，没有问题，收工！很轻易是吧?但是你确认你真正明白了这是怎么回事？数组 b

真的传递到了函数内部？

4.6.1.2，无法向函数传递一个数组

我们完全可以验证一下：

```
void fun(char   a[10])
  {
        int   i = sizeof（a）;
        char c = a[3];
  }
```

如果数组 b 真正传递到函数内部，那 i 的值应该为 10。但是我们测试后发现 i 的值竟然为 4！为什么会这样呢？难道数组 b 真的没有传递到函数内部？是的，确实没有传递过去，这是因为这样一条规则：

C 语言中，当一维数组作为函数参数的时候，编译器总是把它解析成一个指向其首元素首地址的指针。

这么做是有原因的。在 C 语言中，所有非数组形式的数据实参均以传值形式（对实参做一份拷贝并传递给被调用的函数，函数不能修改作为实参的实际变量的值，而只能修改传递给它的那份拷贝）调用。然而，如果要拷贝整个数组，无论在空间上还是在时间上，其开销都是非常大的。更重要的是，在绝大部分情况下，你其实并不需要整个数组的拷贝，你只想告诉函数在那一刻对哪个特定的数组感兴趣。这样的话，为了节省时间和空间，提高程序运行的效率，于是就有了上述的规则。同样的，函数的返回值也不能是一个数组，而只能是指针。这里要明确的一个概念就是：**函数本身是没有类型的，只有函数的返回值才有类型**。很多书都把这点弄错了，甚至出现“XXX 类型的函数”这种说法。简直是荒唐至极！

经过上面的解释，相信你已经理解上述的规定以及它的来由。上面编译器给出的提示，说函数的参数是一个 char*类型的指针，这点相信也可以理解。

既然如此，我们完全可以把 fun 函数改写成下面的样子：

```
void fun(char   *p)
{
    char c = p[3];//或者是  char c = *(p+3);
}
```

同样，你还可以试试这样子：

```
void fun(char   a[10])
{
      char c = a[3];
}
```

```
int main()
{
    char b[100] = "abcdefg";
    fun(b);

    return 0;
}
```

运行完全没有问题。实际传递的数组大小与函数形参指定的数组大小没有关系。既然如此，那我们也可以改写成下面的样子：

```
void fun(char a[ ])
{
    char c = a[3];
}
```

改写成这样或许比较好，至少不会让人误会成只能传递一个 10 个元素的数组。

4.6.2，一级指针参数

4.6.2.1，能否把指针变量本身传递给一个函数

我们把上一节讨论的列子再改写一下：

```
void fun(char *p)
{
    char c = p[3];//或者是 char c = *(p+3);
}

int main()
{
    char *p2 = "abcdefg";
    fun（p2）;
    return 0;
}
```

这个函数调用，真的把 p2 本身传递到了 fun 函数内部吗？

我们知道 p2 是 main 函数内的一个局部变量，它只在 main 函数内部有效。（这里需要澄清一个问题：main 函数内的变量不是全局变量，而是局部变量，只不过它的生命周期和

全局变量一样长而已。全局变量一定是定义在函数外部的。初学者往往弄错这点。）既然它是局部变量，fun 函数肯定无法使用 p2 的真身。那函数调用怎么办？好办：对实参做一份拷贝并传递给被调用的函数。即对 p2 做一份拷贝，假设其拷贝名为_p2。那传递到函数内部的就是_p2 而并非 p2 本身。

4.6.2.2，无法把指针变量本身传递给一个函数

这很像孙悟空拔下一根猴毛变成自己的样子去忽悠小妖怪。所以 fun 函数实际运行时，用到的都是_p2 这个变量而非 p2 本身。如此，我们看下面的例子：

```
void GetMemory（char * p, int num）
{
    p = (char *)malloc(num*sizeof(char));
}


int main()
{
    char *str =  NULL;
    GetMemory（str，10）;
    strcpy(str,”hello”);
    free（str）;//free 并没有起作用，内存泄漏


    return 0;
}
```

在运行 strcpy(str,”hello”)语句的时候发生错误。这时候观察 str 的值，发现仍然为 NULL。也就是说 str 本身并没有改变，我们 malloc 的内存的地址并没有赋给 str，而是赋给了_str。而这个_str 是编译器自动分配和回收的，我们根本就无法使用。所以想这样获取一块内存是不行的。那怎么办? 两个办法：

第一：用 return。

```
char * GetMemory（char * p, int num）
{
    p = (char *)malloc(num*sizeof(char));
    return p；
}


int main()
{
```

```
        char *str =    NULL;
        str = GetMemory（str，10）;
        strcpy(str,”hello”);
        free（str）;

        return 0;
}
```

这个方法简单，容易理解。

第二：用二级指针。

```
void GetMemory（char ** p, int num）
{
        *p = (char *)malloc(num*sizeof(char));
        return p；
}

int main()
{
        char *str =    NULL;
        GetMemory（&str，10）;
        strcpy(str,”hello”);
        free（str）;
        return 0;
}
```

注意，这里的参数是&str 而非 str。这样的话传递过去的是 str 的地址，是一个值。在函数内部，用钥匙（“*”）来开锁：*(&str)，其值就是 str。所以 malloc 分配的内存地址是真正赋值给了 str 本身。

另外关于 malloc 和 free 的具体用法，内存管理那章有详细讨论。

4.6.3，二维数组参数与二维指针参数

前面详细分析了二维数组与二维指针，那它们作为参数时与不作为参数时又有什么区别呢 ？看例子：

```
void fun（char a[3][4]）;
```

我们按照上面的分析，完全可以把 a[3][4]理解为一个一维数组 a[3]，其每个元素都是一个含有 4 个 char 类型数据的数组。上面的规则，“C 语言中，当一维数组作为函数参数的时候，编译器总是把它解析成一个指向其首元素首地址的指针。”在这里同样适用，也就是说我们可以把这个函数声明改写为：

void fun（char (*p)[4]）;

这里的括号绝对不能省略，这样才能保证编译器把 p 解析为一个指向包含 4 个 char 类型数据元素的数组，即一维数组 a[3]的元素。

同样，作为参数时，一维数组“[]”号内的数字完全可以省略：

void fun（char a[ ][4]）;

不过第二维的维数却不可省略，想想为什么不可以省略？

注意：如果把上面提到的声明 void fun（char (*p)[4]）中的括号去掉之后，声明“void f un（char *p[4]）”可以改写成：

void fun（char **p）;

这是因为参数*p[4]，对于 p 来说，它是一个包含 4 个指针的一维数组，同样把这个一维数组也改写为指针的形式，那就得到上面的写法。

上面讨论了这么多，那我们把二维数组参数和二维指针参数的等效关系整理一下：

数组参数	等效的指针参数
数组的数组：char a[3][4]	数组的指针：char (*p)[10]
指针数组： char *a[5]	指针的指针：char **p

这里需要注意的是：**C 语言中，当一维数组作为函数参数的时候，编译器总是把它解析成一个指向其首元素首地址的指针。这条规则并不是递归的，也就是说只有一维数组才是如此，当数组超过一维时，将第一维改写为指向数组首元素首地址的指针之后，后面的维再也不可改写。**比如：a[3][4][5]作为参数时可以被改写为（*p）[4][5]。

至于超过二维的数组和超过二级的指针，由于本身很少使用，而且按照上面的分析方法也能很好的理解，这里就不再详细讨论。有兴趣的可以好好研究研究。

4.7，函数指针

4.7.1，函数指针的定义

顾名思义，函数指针就是函数的指针。它是一个指针，指向一个函数。看例子：

A)，char * (*fun1)(char * p1,char * p2);

B)，char * *fun2(char * p1,char * p2);

C)，char * fun3(char * p1,char * p2);

看看上面三个表达式分别是什么意思？

C）：这很容易，fun3 是函数名，p1，p2 是参数，其类型为 char *型，函数的返回值为 char * 类型。

B)：也很简单，与 C）表达式相比，唯一不同的就是函数的返回值类型为 char**，是个二级指针。

A)：fun1 是函数名吗？回忆一下前面讲解数组指针时的情形。我们说数组指针这么定义或许更清晰：

int (*)[10]　p；

再看看 A）表达式与这里何其相似！明白了吧。这里 fun1 不是什么函数名，而是一个指针变量，它指向一个函数。这个函数有两个指针类型的参数，函数的返回值也是一个指针。同样，我们把这个表达式改写一下：char * (*)(char * p1,char * p2)　fun1; 这样子是不是好看一些呢？只可惜编译器不这么想。^_^。

4.7.2，函数指针的使用

4.7.2.1，函数指针使用的例子

上面我们定义了一个函数指针，但如何来使用它呢？先看如下例子：

```
#include <stdio.h>
#include <string.h>

char * fun(char * p1,char * p2)
{
    int i = 0;
    i = strcmp(p1,p2);
    if (0 == i)
    {
        return p1;
    }
    else
    {
        return p2;
    }
}


int main()
{
    char * (*pf)(char * p1,char * p2);
    pf = &fun;
    (*pf) ("aa","bb");
```

```
    return 0;
}
```

我们使用指针的时候，需要通过钥匙（“*”）来取其指向的内存里面的值，函数指针使用也如此。通过用(*pf)取出存在这个地址上的函数，然后调用它。这里需要注意到是，在 Visual C++6.0 里，给函数指针赋值时，可以用&fun 或直接用函数名 fun。这是因为函数名被编译之后其实就是一个地址，所以这里两种用法没有本质的差别。这个例子很简单，就不再详细讨论了。

4.2.7.2，*(int*)&p ----这是什么？

也许上面的例子过于简单，我们看看下面的例子：

```
void Function()
{
    printf("Call    Function!\n");
}

int main()
{
    void    (*p)();
    *(int*)&p=(int)Function;
    (*p) ();
    return 0;
}
```

这是在干什么？*(int*)&p=(int)Function;表示什么意思？

别急，先看这行代码：

void (*p)();

这行代码定义了一个指针变量 p，p 指向一个函数，这个函数的参数和返回值都是 void。

&p 是求指针变量 p 本身的地址，这是一个 32 位的二进制常数（32 位系统）。

(int*)&p 表示将地址强制转换成指向 int 类型数据的指针。

(int)Function 表示将函数的入口地址强制转换成 int 类型的数据。

分析到这里，相信你已经明白*(int*)&p=(int)Function;表示将函数的入口地址赋值给指针变量 p。

那么(*p) ();就是表示对函数的调用。

讲解到这里，相信你已经明白了。其实函数指针与普通指针没什么差别，只是指向的内容不同而已。

使用函数指针的好处在于，可以将实现同一功能的多个模块统一起来标识，这样一来更容易后期的维护，系统结构更加清晰。或者归纳为：便于分层设计、利于系统抽象、降低耦合度以及使接口与实现分开。

4.7.3，(*(void(*) ())0)()------这是什么？

是不是感觉上面的例子太简单，不够刺激？好，那就来点刺激的，看下面这个例子：

```
(*(void(*) ())0)();
```

这是《C Traps and Pitfalls》这本经典的书中的一个例子。没有发狂吧？下面我们就来分析分析：

第一步：void(*) ()，可以明白这是一个函数指针类型。这个函数没有参数，没有返回值。

第二步：(void(*) ())0，这是将 0 强制转换为函数指针类型，0 是一个地址，也就是说一个函数存在首地址为 0 的一段区域内。

第三步：(*(void(*) ())0)，这是取 0 地址开始的一段内存里面的内容，其内容就是保存在首地址为 0 的一段区域内的函数。

第四步：(*(void(*) ())0)()，这是函数调用。

好像还是很简单是吧，上面的例子再改写改写：

```
(*(char**(*) (char **,char **))0) ( char **,char **);
```

如果没有上面的分析，肯怕不容易把这个表达式看明白吧。不过现在应该是很简单的一件事了。读者以为呢？

4.7.4，函数指针数组

现在我们清楚表达式 “char * (*pf)(char * p)”定义的是一个函数指针 pf。既然 pf 是一个指针，那就可以储存在一个数组里。把上式修改一下：

```
char * (*pf[3])(char * p);
```

这是定义一个函数指针数组。它是一个数组，数组名为 pf，数组内存储了 3 个指向函数的指针。这些指针指向一些返回值类型为指向字符的指针、参数为一个指向字符的指针的函数。这念起来似乎有点拗口。不过不要紧，关键是你明白这是一个指针数组，是数组。

函数指针数组怎么使用呢？这里也给出一个非常简单的例子，只要真正掌握了使用方法，再复杂的问题都可以应对。如下：

```
#include <stdio.h>
#include <string.h>
char * fun1(char * p)
{
    printf("%s\n",p);
    return p;
}

char * fun2(char * p)
{
    printf("%s\n",p);
    return p;
```

```
}

char * fun3(char * p)
{
    printf("%s\n",p);
    return p;
}

int main()
{
    char * (*pf[3])(char * p);
    pf[0] = fun1; // 可以直接用函数名
    pf[1] = &fun2; // 可以用函数名加上取地址符
    pf[2] = &fun3;

   pf[0]("fun1");
   pf[0]("fun2");
   pf[0]("fun3");
   return 0;
}
```

4.7.5，函数指针数组的指针

看着这个标题没发狂吧？函数指针就够一般初学者折腾了，函数指针数组就更加麻烦，现在的函数指针数组指针就更难理解了。

其实，没这么复杂。前面详细讨论过数组指针的问题，这里的函数指针数组指针不就是一个指针嘛。只不过这个指针指向一个数组，这个数组里面存的都是指向函数的指针。仅此而已。

下面就定义一个简单的函数指针数组指针：

```
char * (*(*pf)[3])(char * p);
```

注意，这里的 pf 和上一节的 pf 就完全是两码事了。上一节的 pf 并非指针，而是一个数组名；这里的 pf 确实是实实在在的指针。这个指针指向一个包含了 3 个元素的数组；这个数字里面存的是指向函数的指针；这些指针指向一些返回值类型为指向字符的指针、参数为一个指向字符的指针的函数。这比上一节的函数指针数组更拗口。其实你不用管这么多，明白这是一个指针就 ok 了。其用法与前面讲的数组指针没有差别。下面列一个简单的例子：

```
#include <stdio.h>
#include <string.h>

char * fun1(char * p)
{
    printf("%s\n",p);
    return p;
}

char * fun2(char * p)
{
    printf("%s\n",p);
    return p;
}


char * fun3(char * p)
{
    printf("%s\n",p);
    return p;
}
int main()
{
    char * (*a[3])(char * p);
    char * (*(*pf)[3])(char * p);
    pf = &a;

    a[0] = fun1;
    a[1] = &fun2;
    a[2] = &fun3;

    pf[0][0]("fun1");
    pf[0][1]("fun2");
```

```
    pf[0][2]("fun3");
    return 0;
}
```

第五章 内存管理

欢迎您进入这片雷区。我欣赏能活着走出这片雷区的高手，但更欣赏“粉身碎骨浑不怕，不留地雷在人间”的勇者。请您不要把这当作一个扫雷游戏，因为没有人能以游戏的心态取胜。

曾经很短暂的使用过一段时间的 C#。头三天特别不习惯，因为没有指针！后来用起来越来越顺手，还是因为没有指针！几天的时间很轻易的写了 1 万多行 C#代码，感觉比用 C 或 C++简单多了。因为你根本就不用去考虑底层的内存管理，也不用考虑内存泄漏的问题，更加不怕“野指针”（有的书叫“悬垂指针”）。所有这一切，系统都给你做了，所以可以很轻松的拿来就用。但是 C 或 C++，这一切都必须你自己来处理，即使经验丰富的老手也免不了犯错。我曾经做过一个项目，软件提交给客户很久之后，客户发现一个很严重的 bug。这个 bug 很少出现，但是一旦出现就是致命的，系统无法启动！这个问题交给我来解决。由于要再现这个 bug 十分困难，按照客户给定的操作步骤根本无法再现。经过大概 2 周时间天天和客户越洋视频之后，终于找到了 bug 的原因——野指针！所以关于内存管理，尤其是野指针的问题，千万千万不要掉以轻心，否则，你会很惨的。

5.1，什么是野指针

那到底什么是野指针呢？怎么去理解这个“野”呢？我们先看别的两个关于“野”的词：

野孩子：没人要，没人管的孩子；行为动作不守规矩，调皮捣蛋的孩子。

野狗：没有主人的狗，没有链子锁着的狗，喜欢四处咬人。

对付野孩子的最好办法是给他定一套规矩，好好管教。一旦发现没有按规矩办事就好好收拾他。对付野狗最好的办法就是拿条狗链锁着它，不让它四处乱跑。

对付也指针肯怕比对付野孩子或野狗更困难。我们需要把对付野孩子和野狗的办法都用上。既需要规矩，也需要链子。

前面我们把内存比作尺子，很轻松的理解了内存。尺子上的 0 毫米处就是内存的 0 地址处，也就是 NULL 地址处。这条栓“野指针”的链子就是这个“NULL”。定义指针变量的同时最好初始化为 NULL，用完指针之后也将指针变量的值设置为 NULL。也就是说除了在使用时，别的时间都把指针“栓”到 0 地址处。这样它就老实了。

5.2，栈、堆和静态区

对于程序员，一般来说，我们可以简单的理解为内存分为三个部分：静态区，栈，堆。很多书没有把把堆和栈解释清楚，导致初学者总是分不清楚。其实堆栈就是栈，而不是堆。堆的英文是 heap；栈的英文是 stack，也翻译为堆栈。堆和栈都有自己的特性，这里先不做

讨论。再打个比方：一层教学楼，可能有外语教室，允许外语系学生和老师进入；还可能有数学教师，允许数学系学生和老师进入；还可能有校长办公室，允许校长进入。同样，内存也是这样，内存的三个部分，不是所有的东西都能存进去的。

静态区：保存自动全局变量和 static 变量（包括 static 全局和局部变量）。静态区的内容在总个程序的生命周期内都存在，由编译器在编译的时候分配。

栈：保存局部变量。栈上的内容只在函数的范围内存在，当函数运行结束，这些内容也会自动被销毁。其特点是效率高，但空间大小有限。

堆：由 malloc 系列函数或 new 操作符分配的内存。其生命周期由 free 或 delete 决定。在没有释放之前一直存在，直到程序结束。其特点是使用灵活，空间比较大，但容易出错。

5.3，常见的内存错误及对策

5.3.1，指针没有指向一块合法的内存

定义了指针变量，但是没有为指针分配内存，即指针没有指向一块合法的内存。

浅显的例子就不举了，这里举几个比较隐蔽的例子。

5.3.1.1，结构体成员指针未初始化

```
struct student
{
    char *name;
    int score;
}stu,*pstu;

int main()
{
     strcpy(stu.name,"Jimy");
     stu.score = 99;
     return 0;
 }
```

很多初学者犯了这个错误还不知道是怎么回事。这里定义了结构体变量 stu，但是他没想到这个结构体内部 char *name 这成员在定义结构体变量 stu 时，只是给 name 这个指针变量本身分配了 4 个字节。name 指针并没有指向一个合法的地址，这时候其内部存的只是一些乱码。所以在调用 strcpy 函数时，会将字符串"Jimy"往乱码所指的内存上拷贝，而这块内

存 name 指针根本就无权访问，导致出错。解决的办法是为 name 指针 malloc 一块空间。

同样，也有人犯如下错误：

```
int main()
{
    pstu = (struct student*)malloc(sizeof(struct student));
    strcpy(pstu->name,"Jimy");
    pstu->score = 99;
    free(pstu);

    return 0;
}
```

为指针变量 pstu 分配了内存，但是同样没有给 name 指针分配内存。错误与上面第一种情况一样，解决的办法也一样。这里用了一个 malloc 给人一种错觉，以为也给 name 指针分配了内存。

5.3.1.2，没有为结构体指针分配足够的内存

```
int main()
{
    pstu = (struct student*)malloc(sizeof(struct student*));
    strcpy(pstu->name,"Jimy");
    pstu->score = 99;
    free(pstu);
    return 0;
}
```

为 pstu 分配内存的时候，分配的内存大小不合适。这里把 sizeof(struct student)误写为 sizeof(struct student*)。当然 name 指针同样没有被分配内存。解决办法同上。

5.3.1.3，函数的入口校验

不管什么时候，我们使用指针之前一定要确保指针是有效的。

一般在函数入口处使用 assert(NULL != p)对参数进行校验。在非参数的地方使用 if（NULL != p）来校验。但这都有一个要求，即 p 在定义的同时被初始化为 NULL 了。比如上面的例子，即使用 if（NULL != p）校验也起不了作用，因为 name 指针并没有被初始化为 NULL，其内部是一个非 NULL 的乱码。

assert 是一个宏，而不是函数，包含在 assert.h 头文件中。如果其后面括号里的值为假，则程序终止运行，并提示出错；如果后面括号里的值为真，则继续运行后面的代码。这个宏只在 Debug 版本上起作用，而在 Release 版本被编译器完全优化掉，这样就不会影响代码的性能。

有人也许会问，既然在 Release 版本被编译器完全优化掉，那 Release 版本是不是就完全没有这个参数入口校验了呢？这样的话那不就跟不使用它效果一样吗？

是的，使用 assert 宏的地方在 Release 版本里面确实没有了这些校验。但是我们要知道，assert 宏只是帮助我们调试代码用的，它的一切作用就是让我们尽可能的在调试函数的时候把错误排除掉，而不是等到 Release 之后。它本身并没有除错功能。再有一点就是，参数出现错误并非本函数有问题，而是调用者传过来的实参有问题。assert 宏可以帮助我们定位错误，而不是排除错误。

5.3.2，为指针分配的内存太小

为指针分配了内存，但是内存大小不够，导致出现越界错误。

char *p1 = “abcdefg”;

char *p2 = (char *)malloc(sizeof(char)*strlen(p1));

strcpy(p2,p1);

p1 是字符串常量，其长度为 7 个字符，但其所占内存大小为 8 个 byte。初学者往往忘了字符串常量的结束标志“\0”。这样的话将导致 p1 字符串中最后一个空字符“\0”没有被拷贝到 p2 中。解决的办法是加上这个字符串结束标志符：

char *p2 = (char *)malloc(sizeof(char)*strlen(p1)+1*sizeof(char));

这里需要注意的是，只有字符串常量才有结束标志符。比如下面这种写法就没有结束标志符了：

char a[7] = {‘a’,’b’,’c’,’d’,’e’,’f’,’g’};

另外，不要因为 char 类型大小为 1 个 byte 就省略 sizof（char）这种写法。这样只会使你的代码可移植性下降。

5.3.3，内存分配成功，但并未初始化

犯这个错误往往是由于没有初始化的概念或者是以为内存分配好之后其值自然为 0。未初始化指针变量也许看起来不那么严重，但是它确确实实是个非常严重的问题，而且往往出现这种错误很难找到原因。

曾经有一个学生在写一个 windows 程序时，想调用字库的某个字体。而调用这个字库需要填充一个结构体。他很自然的定义了一个结构体变量，然后把他想要的字库代码赋值给了相关的变量。但是，问题就来了，不管怎么调试，他所需要的这种字体效果总是不出来。我在检查了他的代码之后，没有发现什么问题，于是单步调试。在观察这个结构体变

量的内存时，发现有几个成员的值为乱码。就是其中某一个乱码惹得祸！因为系统会按照这个结构体中的某些特定成员的值去字库中寻找匹配的字体，当这些值与字库中某种字体的某些项匹配时，就调用这种字体。但是很不幸，正是因为这几个乱码，导致没有找到相匹配的字体！因为系统并无法区分什么数据是乱码，什么数据是有效的数据。只要有数据，系统就理所当然的认为它是有效的。

也许这种严重的问题并不多见，但是也绝不能掉以轻心。所以在定义一个变量时，第一件事就是初始化。你可以把它初始化为一个有效的值，比如:

```
int   i = 10；
char *p = (char *)malloc(sizeof(char))；
```

但是往往这个时候我们还不确定这个变量的初值，这样的话可以初始化为 0 或 NULL。

```
int   i = 0；
char *p = NULL；
```

如果定义的是数组的话，可以这样初始化：

```
int a[10] =   {0};
```

或者用 memset 函数来初始化为 0：

```
memset（a,0,sizeof(a)）;
```

memset 函数有三个参数，第一个是要被设置的内存起始地址；第二个参数是要被设置的值；第三个参数是要被设置的内存大小，单位为 byte。这里并不想过多的讨论 memset 函数的用法，如果想了解更多，请参考相关资料。

至于指针变量如果未被初始化，会导致 if 语句或 assert 宏校验失败。这一点，上面已有分析。

5.3.4，内存越界

内存分配成功，且已经初始化，但是操作越过了内存的边界。

这种错误经常是由于操作数组或指针时出现“多 1”或“少 1”。比如：

```
int a[10] = {0};
for (i=0; i<=10; i++)
{
    a[i] = i;
}
```

所以，for 循环的循环变量一定要使用半开半闭的区间，而且如果不是特殊情况，循环变量尽量从 0 开始。

5.3.5，内存泄漏

内存泄漏几乎是很难避免的，不管是老手还是新手，都存在这个问题。甚至包括 windows，Linux 这类软件，都或多或少有内存泄漏。也许对于一般的应用软件来说，这个问题似乎不是那么突出，重启一下也不会造成太大损失。但是如果你开发的是嵌入式系统软件呢？比如汽车制动系统，心脏起搏器等对安全要求非常高的系统。你总不能让心脏起搏器重启吧，人家阎王老爷是非常好客的。

会产生泄漏的内存就是堆上的内存（这里不讨论资源或句柄等泄漏情况），也就是说由 malloc 系列函数或 new 操作符分配的内存。如果用完之后没有及时 free 或 delete，这块内存就无法释放，直到整个程序终止。

5.3.5.1，告老还乡求良田

怎么去理解这个内存分配和释放过程呢？先看下面这段对话：

万岁爷：爱卿，你为朕立下了汗马功劳，想要何赏赐啊？

某功臣：万岁，黄金白银，臣视之如粪土。臣年岁已老，欲告老还乡。臣乞良田千亩以荫后世，别无他求。

万岁爷：爱卿，你劳苦功高，却仅要如此小赏，朕今天就如你所愿。户部刘侍郎，查看湖广一带是否还有千亩上等良田未曾封赏。

刘侍郎：长沙尚有五万余亩上等良田未曾封赏。

万岁爷：在长沙拨良田千亩封赏爱卿。爱卿，良田千亩，你欲何用啊？

某功臣：谢万岁。长沙一带，适合种水稻，臣想用来种水稻。种水稻需要把田分为一亩一块，方便耕种。

。。。。

5.3.5.2，如何使用 malloc 函数

不要莫名其妙，其实上面这段小小的对话，就是 malloc 的使用过程。malloc 是一个函数，专门用来从堆上分配内存。使用 malloc 函数需要几个要求：

内存分配给谁？这里是把良田分配给某功臣。

分配多大内存？这里是分配一千亩。

是否还有足够内存分配？这里是还有足够良田分配。

内存的将用来存储什么格式的数据，即内存用来做什么?这里是用来种水稻，需要把田分成一亩一块。

分配好的内存在哪里？这里是在长沙。

如果这五点都确定，那内存就能分配。下面先看 malloc 函数的原型：

(void *)malloc(int size)

malloc 函数的返回值是一个 void 类型的指针，参数为 int 类型数据，即申请分配的内存大小，单位是 byte。内存分配成功之后，malloc 函数返回这块内存的首地址。你需要一个指针来接收这个地址。但是由于函数的返回值是 void *类型的，所以必须强制转换成你所接收的类型。也就是说，这块内存将要用来存储什么类型的数据。比如：

char *p = (char *)malloc(100);

在堆上分配了 100 个字节内存，返回这块内存的首地址，把地址强制转换成 char *类型后赋给 char *类型的指针变量 p。同时告诉我们这块内存将用来存储 char 类型的数据。也就是说你只能通过指针变量 p 来操作这块内存。这块内存本身并没有名字，对它的访问是匿名访问。

上面就是使用 malloc 函数成功分配一块内存的过程。但是，每次你都能分配成功吗？不一定。上面的对话，皇帝让户部侍郎查询是否还有足够的良田未被分配出去。使用 malloc 函数同样要注意这点：如果所申请的内存块大于目前堆上剩余内存块（整块），则内存分配会失败，函数返回 NULL。注意这里说的“堆上剩余内存块”不是所有剩余内存块之和，因为 malloc 函数申请的是连续的一块内存。

既然 malloc 函数申请内存有不成功的可能，那我们在使用指向这块内存的指针时，必须用 if（NULL ！= p）语句来验证内存确实分配成功了。

5.3.5.3，用 malloc 函数申请 0 字节内存

另外还有一个问题：用 malloc 函数申请 0 字节内存会返回 NULL 指针吗？

可以测试一下，也可以去查找关于 malloc 函数的说明文档。申请 0 字节内存，函数并不返回 NULL，而是返回一个正常的内存地址。但是你却无法使用这块大小为 0 的内存。这好尺子上的某个刻度，刻度本身并没有长度，只有某两个刻度一起才能量出长度。对于这一点一定要小心，因为这时候 if（NULL ！= p）语句校验将不起作用。

5.3.5.4，内存释放

既然有分配，那就必须有释放。不然的话，有限的内存总会用光，而没有释放的内存却在空闲。与 malloc 对应的就是 free 函数了。free 函数只有一个参数，就是所要释放的内存块的首地址。比如上例：

free(p);

free 函数看上去挺狠的，但它到底作了什么呢？其实它就做了一件事：斩断指针变量与这块内存的关系。比如上面的例子，我们可以说 malloc 函数分配的内存块是属于 p 的，因为我们对这块内存的访问都需要通过 p 来进行。free 函数就是把这块内存和 p 之间的所有关系斩断。从此 p 和那块内存之间再无瓜葛。至于指针变量 p 本身保存的地址并没有改变，但是它对这个地址处的那块内存却已经没有所有权了。那块被释放的内存里面保存的值也没有改变，只是再也没有办法使用了。

这就是 free 函数的功能。按照上面的分析，如果对 p 连续两次以上使用 free 函数，肯定会发生错误。因为第一使用 free 函数时，p 所属的内存已经被释放，第二次使用时已经无内存可释放了。关于这点，我上课时让学生记住的是：**一定要一夫一妻制，不然肯定出错。**

malloc 两次只 free 一次会内存泄漏；malloc 一次 free 两次肯定会出错。也就是说，在程序中 malloc 的使用次数一定要和 free 相等，否则必有错误。这种错误主要发生在循环使用 malloc 函数时，往往把 malloc 和 free 次数弄错了。这里留个练习：

写两个函数，一个生成链表，一个释放链表。两个函数的参数都只使用一个表头指针。

5.3.5.5，内存释放之后

既然使用 free 函数之后指针变量 p 本身保存的地址并没有改变，那我们就需要重新把 p 的值变为 NULL：

p = NULL;

这个 NULL 就是我们前面所说的“栓野狗的链子”。如果你不栓起来迟早会出问题的。比如：

在 free（p）之后，你用 if（NULL ！= p）这样的校验语句还能起作用吗？

例如：

```
char *p = (char *) malloc(100);
strcpy(p, “hello”);
free(p);        /* p 所指的内存被释放，但是 p 所指的地址仍然不变 */
…
if (NULL != p)
{
        /* 没有起到防错作用 */
        strcpy(p, “world”);     /* 出错 */
}
```

释放完块内存之后，没有把指针置 NULL，这个指针就成为了“野指针”，也有书叫“悬垂指针”。这是很危险的，而且也是经常出错的地方。所以一定要记住一条：free 完之后，一定要给指针置 NULL。

同时留一个问题：对 NULL 指针连续 free 多次会出错吗?为什么？如果让你来设计 free 函数，你会怎么处理这个问题？

5.3.6，内存已经被释放了，但是继续通过指针来使用

这里一般有三种情况：

第一种：就是上面所说的，free（p）之后，继续通过 p 指针来访问内存。解决的办法就是给 p 置 NULL。

第二种：函数返回栈内存。这是初学者最容易犯的错误。比如在函数内部定义了一个数组，却用 return 语句返回指向该数组的指针。解决的办法就是弄明白栈上变量的生命周期。

第三种：内存使用太复杂，弄不清到底哪块内存被释放，哪块没有被释放。解决的办法是重新设计程序，改善对象之间的调用关系。

上面详细讨论了常见的六种错误及解决对策，希望读者仔细研读，尽量使自己对每种错误发生的原因及预防手段烂熟于胸。一定要多练，多调试代码，同时多总结经验。

第六章 函数

什么是函数？为什么需要函数？这两个看似很简单的问题，你能回答清楚吗？

6.1，函数的由来与好处

其实在汇编语言阶段，函数这个概念还是比较模糊的。汇编语言的代码往往就是从入口开始一条一条执行，直到遇到跳转指令（比如 ARM 指令 B、BL、BX、BLX 之类）然后才跳转到目的指令处执行。这个时候所有的代码仅仅是按其将要执行的顺序排列而已。后来人们发现这样写代码非常费劲，容易出错，也不方便。于是想出一个办法，把一些功能相对来说能成为一个整体的代码放到一起打包，通过一些数据接口和外界通信。这就是函数的由来。那函数能给我们带来什么好处呢？简单来说可以概括成以下几点：

1、降低复杂性：使用函数的最首要原因是为了降低程序的复杂性，可以使用函数来隐含信息，从而使你不必再考虑这些信息。

2、避免重复代码段：如果在两个不同函数中的代码很相似，这往往意味着分解工作有误。这时，应该把两个函数中重复的代码都取出来，把公共代码放入一个新的通用函数中，然后再让这两个函数调用新的通用函数。通过使公共代码只出现一次，可以节约许多空间。因为只要在一个地方改动代码就可以了。这时代码也更可靠了。

3、限制改动带来的影响：由于在独立区域进行改动，因此，由此带来的影响也只限于一个 或最多几个区域中。

4、隐含顺序：如果程序通常先从用户那里读取数据，然后再从一个文件中读取辅助数据，在设计系统时编写一个函数，隐含哪一个首先执行的信息。

5、改进性能：把代码段放入函数也使得用更快的算法或执行更快的语言（如汇编）来改进这段代码的工作变得容易些。

6、进行集中控制：专门化的函数去读取和改变内部数据内容，也是一种集中的控制形式。

7、隐含数据结构：可以把数据结构的实现细节隐含起来。

8、隐含指针操作：指针操作可读性很差，而且很容易引发错误。通过把它们独立在函数中，可以把注意力集中到操作意图而不是集中到的指针操作本身。

9、隐含全局变量：参数传递。

C 语言中，函数其实就是一些语句的的集合，而语句又是由关键字和符号等元素组成，如果我们把关键字、符号等基本元素弄明白了，函数不就没有问题了么？我看未必。真正要编写出高质量的函数来，是非常不容易的。前辈们经过大量的探讨和研究总结出来一下一些通用的规则和建议：

6.2，编码风格

很多人不重视这点，认为无所谓，甚至国内的绝大多数教材也不讨论这个话题，导致学

生入公司后仍要进行编码风格的教育。我接触过很多学生，发现他们由于平时缺乏这种意识，养成了不好的习惯，导致很难改正过来。代码没有注释，变量、函数等命名混乱，过两天自己都看不懂自己的代码。下面是一些我见过的比较好的做法，希望读者能有所收获。

【规则6-1】每一个函数都必须有注释，即使函数短到可能只有几行。头部说明需要包含包含的内容和次序如下：

```
/************************************************************************
*     Function Name                    :     nucFindThread
*     Create Date                      :     2000/01/07
*     Author/Corporation               :     your name/your company name
*
*     Description                            :     Find a proper thread in thread array.
*                                              If it's a new then search an empty.
*
*     Param                            :     ThreadNo：   someParam description
*                                              ThreadStatus：  someParam description
*
*     Return Code                      :     Return Code description,eg:
                                               ERROR_Fail: not find a thread
                                               ERROR_SUCCEED: found
*
*     Global Variable                  :     DISP_wuiSegmentAppID
*     File Static Variable             :     naucThreadNo
*     Function Static Variable         :     None
*
*------------------------------------------------------------------------
*     Revision History
*     No.        Date           Revised by        Item          Description
*     V0.5       2008/01/07     your name         …             …
************************************************************************/
static unsigned char nucFindThread(unsigned char ThreadNo,unsigned char ThreadStatus)
{
      …
}
```

【规则6-2】每个函数定义结束之后以及每个文件结束之后都要加一个或若干个空行。例如：

```
/************************************************************************
*     ………
* Function1 Description
*     ………
************************************************************************/
void Function1(……)
{
    …
}
                                              //Blank Line
/************************************************************************
*     ………
* Function2 Description
*     ………
************************************************************************/
void Function2(……)
{
    …
}
                                              //Blank Line
/************************************************************************
*     ………
* Function3 Description
```

```
*       .........
*************************************************************************/
void Function3(……)
{
     …
}
                                           //Blank Line

```

【规则6-3】在一个函数体内，变量定义与函数语句之间要加空行。

例如：

```
/*************************************************************************
*       .........
* Function Description
*.........
*************************************************************************/
void Function1()
{
        int n;
               //Blank Line
        statement1
        …….
}
```

【规则6-4】逻辑上密切相关的语句之间不加空行，其它地方应加空行分隔。

例如：

```
               //Blank Line
        while (condition)
               {
                statement1;
                        //Blank Line
                if (condition)
                        {
                        statement2;
                }
                        else
                        {
                        statement3;
                }
                        //Blank Line
                statement4
        }
```

【规则6-5】复杂的函数中，在分支语句，循环语句结束之后需要适当的注释，方便区分各分支或循环体

```
                        while (condition)
                        {
                                statement1;

                                if (condition)
                                {
                                        for(condition)
                                        {
                                            Statement2;
                                        }//end “for(condition)”
                                }
                                else
                                {
```

```
            statement3;
        }//"end if (condition)"

        statement4
    }//end "while (condition)"
```

【规则6-6】修改别人代码的时候不要轻易删除别人的代码，应该用适当的注释方式，例如：

```
while (condition)
        {
        statement1;

       /////////////////////////////////////
        //your name , 2008/01/07 delete
        //if (condition)
        //{
        //    for(condition)
        //    {
        //            Statement2;
        //    }
        //}
        //else
        //{
        //        statement3;
        //}
        ////////////////////////////////////////

        ///////////////////////////////////////
        // your name , 2000/01/07    add
                …
                new code
                …
      ///////////////////////////////////////

        statement4
}
```

【规则6-7】用缩行显示程序结构，使排版整齐，缩进量统一使用4个字符（不使用TAB缩进）。

每个编辑器的TAB键定义的空格数不一致，可能导致在别的编辑器打开你的代码乱成一团糟。

【规则6-8】在函数体的开始、结构/联合的定义、枚举的定义以及循环、判断等语句中的代码都要采用缩行。

【规则6-9】同层次的代码在同层次的缩进层上。

例如：

提倡的的风格	不提倡的风格

void Function(int x)
{
        //program code
}
	void Function(int x)
	{
	//program code
	}

struct tagMyStruct	struct tagMyStruct{

{ int a; int b; int c; };	int a; int b; int c; };
if (condition) { //program code } else { //program code }	if (condition){ //program code }else{ //program code }

【规则6-10】代码行最大长度宜控制在80 个字符以内，较长的语句、表达式等要分成多行书写。

【规则6-11】长表达式要在低优先级操作符处划分新行，操作符放在新行之首（以便突出操作符）。拆分出的新行要进行适当的缩进，使排版整齐，语句可读。

例如：

```
if ((very_longer_variable1 >= very_longer_variable12)
        && (very_longer_variable3 <= very_longer_variable14)
        && (very_longer_variable5 <= very_longer_variable16))
{
            dosomething();
}

for (very_longer_initialization;
          very_longer_condition;
          very_longer_update)
{
        dosomething();
}
```

【规则6-12】如果函数中的参数较长，则要进行适当的划分。

例如：

```
void function(float very_longer_var1,
                        float very_longer_var2,
                        float very_longer_var3)
```

【规则6-13】用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。

例如：

```
int aiMinValue;
int aiMaxValue;
int niSet_Value(…);
int niGet_Value(…);
```

【规则6-14】如果代码行中的运算符比较多，用括号确定表达式的操作顺序，避免使用默认的优先级。

例如：

```
leap_year = ((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0);
```

【规则6-15】不要编写太复杂的复合表达式。

例如：

```
i = a >= b && c < d && c + f <= g + h;
```
复合表达式过于复杂

【规则6-16】不要有多用途的复合表达式。

例如：

```
d = (a = b + c) + r;
```

该表达式既求a 值又求d 值。应该拆分为两个独立的语句：

```
a = b + c;
d = a + r;
```

【建议6-17】尽量避免含有否定运算的条件表达式。

例如：

如： `if (!(num >= 10))`

应改为： `if (num < 10)`

【规则6-18】参数的书写要完整，不要贪图省事只写参数的类型而省略参数名字。如果函数没有参数，则用void 填充。

例如：

提倡的风格	不提倡的风格
void set_value(int width, int height); float get_value(void);	void set_value (int, int); float get_value ();

6.2, 函数设计的一般原则和技巧

【规则6-19】原则上尽量少使用全局变量，因为全局变量的生命周期太长，容易出错，也会长时间占用空间. 各个源文件负责本身文件的全局变量, 同时提供一对对外函数, 方便其它函数使用该函数来访问变量。比如：niSet_ValueName(…)；niGet_ValueName(…)；不要直接读写全局变量，尤其是在多线程编程时，必须使用这种方式，并且对读写操作加锁。

【规则6-20】参数命名要恰当，顺序要合理。

例如编写字符串拷贝函数str_copy，它有两个参数。如果把参数名字起为str1 和str2，例如

```
void str_copy (char *str1, char *str2);
```

那么我们很难搞清楚究竟是把str1 拷贝到str2 中，还是刚好倒过来。

可以把参数名字起得更有意义，如叫strSource 和strDestination。这样从名字上就可以看出应该把strSource 拷贝到strDestination。

还有一个问题，这两个参数那一个该在前那一个该在后？参数的顺序要遵循程序员的习惯。一般地，应将目的参数放在前面，源参数放在后面。

如果将函数声明为：

```
void str_copy (char *strSource, char *strDestination);
```

别人在使用时可能会不假思索地写成如下形式：

```
char str[20];
str_copy (str, "Hello World");  参数顺序颠倒
```

【规则6-21】如果参数是指针，且仅作输入参数用，则应在类型前加const，以防止该指针在函数体内被意外修改。

例如：

```
void str_copy (char *strDestination， const char *strSource);
```

【规则6-22】不要省略返回值的类型，如果函数没有返回值，那么应声明为void 类型。如果没有返回值，编译器则默认为函数的返回值是int类型的。

【规则6-23】在函数体的“入口处”，对参数的有效性进行检查。尤其是指针参数，尽量使用assert宏做入口校验，而不使用if语句校验。（关于此问题讨论，详见指针与数组那章。）

【规则6-24】return 语句不可返回指向“栈内存”的“指针”，因为该内存在函数体结束时被自动销毁。例如：

```
char * Func(void)
{
    char str[30];
    …
    return str;
}
```

str 属于局部变量，位于栈内存中，在Func 结束的时候被释放，所以返回str 将导致错误。

【规则6-25】函数的功能要单一，不要设计多用途的函数。微软的Win32 API就是违反本规则的典型，其函数往往因为参数不一样而功能不一，导致很多初学者迷惑。

【规则6-26】函数体的规模要小，尽量控制在80 行代码之内。

【建议6-27】相同的输入应当产生相同的输出。尽量避免函数带有“记忆”功能。

带有“记忆”功能的函数，其行为可能是不可预测的，因为它的行为可能取决于某种“记忆状态“。这样的函数既不易理解又不利于测试和维护。在C 语言中，函数的static局部变量是函数的“记忆”存储器。建议尽量少用static 局部变量，除非必需。

【建议6-28】避免函数有太多的参数，参数个数尽量控制在4个或4个以内。如果参数太多，在使用时容易将参数类型或顺序搞错。微软的Win32 API就是违反本规则的典型，其函数的参数往往七八个甚至十余个。比如一个CreateWindow函数的参数就达11个之多。

【建议6-29】尽量不要使用类型和数目不确定的参数。

C 标准库函数printf 是采用不确定参数的典型代表，其原型为：

```
int printf(const chat *format[, argument]…);
```

这种风格的函数在编译时丧失了严格的类型安全检查。

【建议6-30】有时候函数不需要返回值，但为了增加灵活性如支持链式表达，可以附加返回值。例如字符串拷贝函数strcpy 的原型：

```
char *strcpy(char *strDest，const char *strSrc);
```

strcpy 函数将strSrc 拷贝至输出参数strDest 中，同时函数的返回值又是strDest。这样做并非多此一举，可以获得如下灵活性：

```
char str[20];
int length = strlen(strcpy(str, “Hello World”) );
```

【建议6-31】不仅要检查输入参数的有效性，还要检查通过其它途径进入函数体内的变量的有效性，例如全局变量、文件句柄等。

【规则6-32】函数名与返回值类型在语义上不可冲突。

违反这条规则的典型代表就是C语言标准库函数getchar。几乎没有一部名著没有提到getchar函数，因为它实在太经典，太容易让人犯错误了。所以，每一个有经验的作者都会拿这个例子来警示他的读者，我这里也是如此：

```
char c;
c = getchar();
if(EOF == c)
{
    …
}
```

按照 getchar 名字的意思，应该将变量 c 定义为 char 类型。但是很不幸，getchar 函数的返回值却是 int 类型，其原型为：

```
int getchar(void);
```

由于 c 是 char 类型的，取值范围是[-128,127],如果宏 EOF 的值在 char 的取值范围之外，EOF 的值将无法全部保存到 c 内，会发生截断，将 EOF 值的低 8 位保存到 c 里。这样 if 语句有可能总是失败。这种潜在的危险，如果不是犯过一次错，肯怕很难发现。

6.4，函数递归

6.4.1，一个简单但易出错的递归例子

几乎每一本 C 语言基础的书都讲到了函数递归的问题，但是初学者仍然容易在这个地方犯错误。先看看下面的例子：

```
void fun(int i)
{
    if (i>0)
    {
        fun(i/2);
    }
    printf("%d\n",i);
}

int main()
```

```
{
    fun(10);

    return 0;
}
```
问：输出结果是什么？

这是我上课时，一个学生问我的问题。他不明白为什么输出的结果会是这样：

0

1

2

5

10

他认为应该输出 0。因为当 i 小于或等于 0 时递归调用结束，然后执行 printf 函数打印 i 的值。

这就是典型的没明白什么是递归。其实很简单，printf("%d\n",i);语句是 fun 函数的一部分，肯定执行一次 fun 函数，就要打印一行。怎么可能只打印一次呢？关键就是不明白怎么展开递归函数。展开过程如下：

```
void fun(int i)
{
    if (i>0)
    {
        //fun(i/2);
        if(i/2>0)
        {
            if(i/4>0)
            {
                …
            }
            printf("%d\n",i/4);
        }
        printf("%d\n",i/2);
    }
    printf("%d\n",i);
}
```
这样一展开，是不是清晰多了？其实递归本身并没有什么难处，关键是其展开过程别弄错了。

6.4.2，不使用任何变量编写 strlen 函数

看到这里，也许有人会说，strlen 函数这么简单，有什么好讨论的。是的，我相信你能熟练应用这个函数，也相信你能轻易的写出这个函数。但是如果我把要求提高一些呢：

不允许调用库函数，也不允许使用任何全局或局部变量编写 int my_strlen (char *strDest);

似乎问题就没有那么简单了吧？这个问题曾经在网络上讨论的比较热烈，我几乎是全程“观战”，差点也忍不住手痒了。不过因为我的解决办法在我看到帖子时已经有人提出了，所以作罢。

解决这个问题的办法由好几种，比如嵌套有编语言。因为嵌套汇编一般只在嵌入式底层开发中用到，所以本书就不打算讨论 C 语言嵌套汇编的知识了。 有兴趣的读者，可以查找相关资料。

也许有的读者想到了用递归函数来解决这个问题。是的，你应该想得到，因为我把这个问题放在讲解函数递归的时候讨论。既然已经有了思路，这个问题就很简单了。代码如下：

```
int my_strlen( const char* strDest )
{
    assert(NULL != strDest);
    if ('\0' == *strDest)
    {
        return 0;
    }
    else
    {
        return (1 + my_strlen(++strDest));
    }
}
```

第一步：用 assert 宏做入口校验。

第二步：确定参数传递过来的地址上的内存存储的是否为'\0'。如果是，表明这是一个空字符串，或者是字符串的结束标志。

第三步：如果参数传递过来的地址上的内存不为'\0'，则说明这个地址上的内存上存储的是一个字符。既然这个地址上存储了一个字符，那就计数为 1，然后将地址加 1 个 char 类型元素的大小，然后再调用函数本身。如此循环，当地址加到字符串的结束标志符'\0'时，递归停止。

当然，同样是利用递归，还有人写出了更加简洁的代码：

```
int my_strlen( const char* strDest )
{
    return *strDest?1+strlen(strDest+1):0;
}
```

这里很巧妙的利用了问号表达式，但是没有做参数入口校验，同时用*strDest 来代替('\0' == *strDest)也不是很好。所以，这种写法虽然很简洁，但不符合我们前面所讲的编码规范。可以改写一下：

```
int my_strlen( const char* strDest )
{
    assert(NULL != strDest);
    return ('\0' != *strDest)?(1+my_strlen(strDest+1)):0;
}
```

上面的问题利用函数递归的特性就轻易的搞定了，也就是说每调用一遍 my_strlen 函数，其实只判断了一个字节上的内容。但是，如果传入的字符串很长的话，就需要连续多次函数调用，而函数调用的开销比循环来说要大得多，所以，递归的效率很低，递归的深度太大甚至可能出现错误（比如栈溢出）。所以，平时写代码，不到万不得已，尽量不要用递归。即

便是要用递归，也要注意递归的层次不要太深，防止出现栈溢出的错误；同时递归的停止条件一定要正确，否则，递归可能没完没了。

第七章　文件结构

一个工程是往往由多个文件组成。这些文件怎么管理、怎么命名都是非常重要的。下面给出一些基本的方法，比较好的管理这些文件，避免错误的发生。

7.1，文件内容的一般规则

【规则7-1】每个头文件和源文件的头部必须包含文件头部说明和修改记录。

源文件和头文件的头部说明必须包含的内容和次序如下：

```
/**********************************************************************
*    File Name               :    FN_FileName.c/ FN_FileName.h
*    Copyright               :    2003-2008 XXXX Corporation, All Rights Reserved.
*    Module Name             :    Draw Engine/Display
*
*    CPU                     :    ARM7
*    RTOS                    :    Tron
*
*    Create Date             :    2008/10/01
*    Author/Corporation      :    WhoAmI/your company name
*
*    Abstract Description    :    Place some description here.
*
*-----------------------Revision History--------------------------------
*    No   Version      Date       Revised By      Item           Description
*    1    V0.95        08.05.18   WhoAmI       abcdefghijklm       WhatUDo
*
**********************************************************************/
```

【规则7-2】各个源文件必须有一个头文件说明，头文件各部分的书写顺序下：

No.	Item
1	Header File Header Section
2	Multi-Include-Prevent Section
3	Debug Switch Section
4	Include File Section
5	Macro Define Section
6	Structure Define Section
7	Prototype Declare Section

其中 Multi-Include-Prevent Section 是用来防止头文件被重复包含的。

如下例：

```
#ifndef __FN_FILENAME_H
#define __FN_FILENAME_H
#endif
```

其中“FN_FILENAME”一般为本头文件名大写，这样可以有效避免重复，因为同一工程中不可能存在两个同名的头文件。

```
/**************************************************************************
*     File Name                    :     FN_FileName.h
*     Copyright                    :     2003-2008 XXXX Corporation, All Rights Reserved.
*     Module Name                  :     Draw Engine/Display
*
*     CPU                          :     ARM7
*     RTOS                         :     Tron
*
*     Create Date                  :     2008/10/01
*     Author/Corporation           :     WhoAmI/your company name
*
*     Abstract Description         :     Place some description here.
*
*----------------------------------------Revision History---------------------------------
*     No   Version       Date       Revised By       Item            Description
*     1    V0.95         08.05.18   WhoAmI       abcdefghijklm        WhatUDo
*
**************************************************************************/
/**************************************************************************
*       Multi-Include-Prevent Section
**************************************************************************/
#ifndef    __FN_FILENAME_H
#define    __FN_FILENAME_H

/**************************************************************************
*       Debug switch Section
**************************************************************************/
#define    D_DISP_BASE

/**************************************************************************
*       Include File Section
**************************************************************************/
#include  "IncFile.h"

/**************************************************************************
*       Macro Define Section
**************************************************************************/
#define    MAX_TIMER_OUT            (4)

/**************************************************************************
*       Struct    Define Section
**************************************************************************/
typedef struct CM_RadiationDose
{
unsigned char ucCtgID;
      char cPatId_a[MAX_PATI_LEN];
}CM_RadiationDose_st, *CM_RadiationDose_pst;

/**************************************************************************
*       Prototype Declare Section
**************************************************************************/
unsigned int MD_guiGetScanTimes(void);
…
…
#endif
```

【规则7-3】源文件各部分的书写顺序如下：

No.	Item
1	Source File Header Section
2	Debug Switch Section
3	Include File Section
4	Macro Define Section
5	Structure Define Section
6	Prototype Declare Section
7	Global Variable Declare Section
8	File Static Variable Define Section
9	Function Define Section

```
/**************************************************************************
*    File Name              :    FN_FileName.c
*    Copyright              :    2003-2008 XXXX Corporation, All Rights Reserved.
*    Module Name            :    Draw Engine/Display
*
*    CPU                    :    ARM7
*    RTOS                   :    Tron
*
*    Create Date            :    2003/10/01
*    Author/Corporation     :    WhoAmI/your company name
*
*    Abstract Description   :    Place some description here.
*
*-----------------------Revision History--------------------------------
*    No   Version    Date       Revised By    Item           Description
*    1    V0.95      00.05.18   WhoAmI        abcdefghijklm  WhatUDo
*
**************************************************************************/


/**************************************************************************
*      Debug switch Section
**************************************************************************/
#define    D_DISP_BASE


/**************************************************************************
*      Include File Section
**************************************************************************/
#include  "IncFile.h"


/**************************************************************************
*      Macro Define Section
**************************************************************************/
#define    MAX_TIMER_OUT          (4)


/**************************************************************************
```

```
*       Struct    Define Section
**************************************************************************/
typedef struct CM_RadiationDose
{
unsigned char ucCtgID;
        char cPatId_a[MAX_PATI_LEN];
}CM_RadiationDose_st, *pCM_RadiationDose_st;

/*************************************************************************
*       Prototype Declare Section
**************************************************************************/
unsigned int MD_guiGetScanTimes(void);

/*************************************************************************
*       Global Variable Declare Section
**************************************************************************/
extern unsigned int MD_guiHoldBreathStatus;

/*************************************************************************
*       File Static Variable Define Section
**************************************************************************/
static unsigned int nuiNaviSysStatus;

/*************************************************************************
*       Function Define Section
**************************************************************************/
```

【规则7-4】需要对外公开的常量放在头文件中，不需要对外公开的常量放在定义文件的头部。

7.2，文件名命名的规则

【规则7-5】文件标识符分为两部分，即文件名前缀和后缀。文件名前缀的最前面要使用范围限定符——模块名（文件名）缩写；

【规则7-6】采用小写字母命名文件，避免使用一些比较通俗的文件名，如：public.c 等。